$24.95

Basic TIPS

by AMLIST

Comprehensive Programming Instructions For The TI-99/4A Home Computer

Basic TIPS

by AMLIST

This book was authored by: Terrance K. Castle.

Mr. Castle holds a BS degree in Business Administration from Rochester Institute of Technology, Rochester, NY. He has been actively programming microcomputers since 1977, and has developed entire business packages including packages for accounts payable/receivable, general ledger, journal, job costing, and payroll. Prior to his involvement with the TI-99/4A, most of his microcomputer experience was on a Polymorphic 8813 with Mass Storage Unit (2.5 mg). He is familiar with disk drive, multi user, and data base applications.

Acknowledgement:

Many thanks to Larry Z. Isaacson for his editorial assistance and his unfailing support of this effort; and to George H. Faulkner for making the opportunity available.

Published by:

AMLIST, Inc., 4542 Memorial Drive, Suite 202, Atlanta, GA 30032
Telephone: (404) 292-0576

FOREWORD

You entered the fascinating world of micro computers when you purchased your TI-99/4A. You probably already have in mind some very specific uses for it, such as: checkbook management, creative games, or educating yourself and your children. We can assure you you've made a wise choice with this system. TI's instructional material, their documentation, and their support is without equal in the industry and they do provide the new user with a sound foundation on which to build.

Buying command modules or pre-programmed cassettes is a part of this world and certainly makes the computer far more useful. For many, it's the only thing they'll ever need to benefit from their new "in house" computer. However, the real joy and challenge of owning a computer is getting the machine to do what you want it to do -- making it an extension of your own thoughts.

"Basic TIPS" culminates many months of effort to understand the needs of the new computer owner. It effectively answers the most commonly asked questions and, at the same time, presents sophisticated techniques for use by the more experienced programmers. You'll need only console basic and one cassette recorder to operate any of the programs or perform any of the examples in this manual. However, the knowledge you gain and techniques you learn will carry forward regardless of how far you expand your system.

The instructional material herein was initially offered as part of a twelve part series. During that time the participants were supported with a nationwide toll free WATS lines. With the thousands of phone calls received, we've had an opportunity to listen to the needs of the growing number of computer owners, both new and reasonably experienced. We've helped them when possible over the phone, and we've talked them through debugging of programs from whatever source. We've surveyed our readers as to their thoughts about the subject matter and its organization. This finished manual is, in large part, a product of that valuable feedback. We feel abundantly confident that it will answer the needs of any computer owner who sincerely wants to learn more about the art of programming. The fact that it is now available as one complete manual does warrant a word of caution in its use.

In order to obtain maximum benefit from this material, each individual chapter requires thought, study, and hands on effort. You should not expect to complete it within a week or even a month of part time work. A quick scanning of the manual and what it includes would be wise and, to answer a specific point, you can refer to the appropriate chapter; however, having done that, you are encouraged to return to where you left off and proceed forward again from that point. Truly experienced programmers may find it convenient to read the first three

chapters lightly and concentrate more fully on later sections. For those who are not at this level, an attempt to understand the material presented under Data files, for example, without the background provided by the earlier chapters, may prove difficult.

In many instances we'll ask you to type in examples which we will then discuss in detail. Early in the manual we encourage you to enter some of the examples in the manual you received with your computer. You'll learn far more by seeing these in action on the screen than you will by simply reading about them. Repetition is the key to memory retention and the easiest and quickest way to learn commands is to type them often. It is not the purpose of this manual to teach you the commands; rather, it is our goal to teach you how to use them effectively. Most of the programs included herein are not short example type programs. While some are fairly short, others are, in terms of lines of code, quite long. The programs are what they need to be in order to perform their respective tasks. They were not randomly selected from outside sources -- they were specifically designed to enhance the learning process and to illustrate specific points. For the most part, the programs are presented in conjunction with specific chapters to illustrate particular techniques and they are written with a progressively higher level of sophistication. For comparison purposes, you might want to look at the Building Blocks program as it was offered in Chapter 2 compared to the version shown in Chapter 10. Through the use of the techniques taught in this manual, the number of lines of code has been reduced from 448 lines to 129 -- a reduction of more than two thirds. Hopefully, each of you will find at least four or five of these programs particularly useful or entertaining from your viewpoint. Even if you have no need for a specific program, you are encouraged to enter it, read the accompanying write up, and review it if reference is made to it in the instructional part of the manual.

You may discover after the first few chapters that programming is not your "cup of tea". Contrary to popular belief, serious programs and complex games are not just "jotted off" in a few leisurely hours at the keyboard. Even for experienced programmers, they take time and patience and often many hours of trial and error. If you discover this early on, prior to investing a lot of money in peripherals and expansions, this manual will still have served its purpose.

Others of you will find that you have a real knack for this type of thinking. You'll be amazed at the power of this electronic marvel and the amount of information and entertainment it can provide. The problem solving approach, and the probing and testing procedures used herein, will enable you to step into each new expansion with confidence, so that you can get the absolute maximum out of each addition to your system.

We wish you ---

HAPPY COMPUTING!

AMLIST, Inc.

T A B L E   O F   C O N T E N T S

CHAPTER ONE

Introduction & Manual Review

GENERAL. Until recent years, computers were purchased primarily, in fact almost exclusively, by individuals who, when they purchased it, already had at least some knowledge about data processing and computers. In most cases these individuals had spent numerous hours reading data processing or computer oriented magazines and other material prior to making a purchase. They generally knew what they wanted to do with the computer; they knew the strengths and weaknesses of various computers; and they were aware of the differing languages available such as COBAL, FORTRAN, PASCAL, BASIC, etc. In spite of their knowledge of computers and data processing, these first time buyers were still considered "beginners". Instruction manuals, articles, and publications were written to their level of understanding. Thanks to modern technology and mass media advertising, the word "beginner" has taken on a new meaning.

Today's home computer buyer isn't just the ambitious, white collar businessman of a few years ago. We know, from hundreds of discussions with new owners, that the new breed may be a blue collar worker, teacher, housewife, or any person from any walk of life. Some of these people realize that a basic understanding of these seemingly complex machines is necessary to financial survival in this modern age. Their very jobs may depend on their understanding of computers and data processing. Others have simply been caught up in the excitement of the computer age and have bought it because others in their peer group have become involved and are now talking about "programming" their home computers. Many see it as a necessity for their children. They hope that by having a computer readily available that their offspring will be better able to compete in the job market of the future than perhaps they were.

Whatever your reason may be, you've probably come to realize that understanding computers involves more than being able to plug in a "Command Module". When you graduated from a game machine, with its switches and joysticks, to a computer with a functional keyboard, you took a giant step forward. Computers are truly miraculous machines, capable of doing not just what the manufacturer decided it can do, but capable of actually doing what you want it to do when properly programmed. This manual is devoted to those of you who actually want to learn how to program.

Programming. When you have entered the basic language, typed in a line number, and instructed the computer to perform a task, you've written a program. A simple statement such as:

```
>10 PRINT 1+2
>RUN
```

is technically a program, and what you've done is programming. However, having accomplished this, most people wouldn't necessarily consider themselves programmers.

What most people think of as a program is actually what's known as a "user friendly" program. It's a program which, once it is started (RUN), requires no knowledge of the computer or the BASIC language to operate or run. It impresses the neighbors, the kids, and the wife. It puts "neat" things on the screen, asks the user to answer simple questions, or allows the user to move things around with joysticks or up and down keys. It produces the screen display you see at the hardware store that accepts a simple stock number and then displays the item, price, total invoice, sales tax, etc. and actually prints out the sales ticket. Programming is the act of coding in the lines necessary to produce this "user friendly" program. "Program Design" is the act of thinking up what type of information the program accepts and what it's supposed to do with that information. When you come up with the idea to build a file of your personal monthly checks so that you can later use that file to balance your checkbook or compare it with a budget, you've actually gotten into the area of "Systems Design". In other words, you've designed a series of programs that work hand in hand to accomplish some end. In the business world, these jobs are normally performed by entirely different people and the skills necessary to accomplish each task vary greatly. In the home, the "programmer" generally wears all three hats.

Console Basic. Every program and every subroutine in this manual was designed strictly for the built-in Console Basic on the TI-99/4A Home Computer. The only peripheral (other piece of equipment) necessary for completion is one cassette recorder. While the addition of a second recorder, printer, additional memory, disk drive, or Extended Basic, would certainly add to the power of the computer, they will not, in themselves, make learning any easier. We don't give our children electronic calculators in the second grade to learn math and we don't just hand them a dictionary to learn how to spell. The step-by-step process, fewer commands, and limited memory available with Console Basic will make you appreciate the value of these additions much more when they're eventually acquired. The purpose of including programs in this manual is to give you a feel for what can and cannot be done with Console Basic. For some, who purchased the TI-99/4A with a specific use in mind, the limitations may come as a disappointment. For others, you may be pleasantly surprised to find out just how powerful and useful it can be.

Great Expectations. Whether this manual makes it possible for you to design and create your own programs or not will depend largely on how seriously you dedicate yourself to the task of learning. Extensive time will not be spent on teaching you the commands available in the BASIC language or the proper way to type in these commands. From a technical standpoint, these are well documented and covered in the instructional manuals that came with your computer. If you have an aptitude and interest in programming then you'll find this manual extremely helpful in deciding whether your ideas are practical and in converting those ideas to completed programs.

Almost anyone, if they type in all of the programs in this manual, will know how to code in programs. That is, they'll know all of the commands

available in Console Basic, from repetition alone; they'll know how to enter them into the computer; and they'll be able to get the computer to run. However, repetition alone will not make you a "program designer" or "systems designer". This is a thought process that depends on your ability to think out the problem or goal and to logically develop a plan to solve the problem or reach the goal. Just because you learn to read sheet music, or learn to strike a chord on the piano, you don't automatically become a concert pianist, much less a composer. A great deal of time and practice is necessary to accomplish these ends. Programming doesn't take a lifetime to learn, but like anything else worthwhile, it does take practice. If you have the basic skills, a sincere interest, and you apply yourself regularly to the task, within a few months you will not only be able to copy written programs; but, you'll be able to modify and rewrite programs written for other computers, and you'll be able accomplish many other things within the limitations of the system.

Although we've indicated that it takes patience, you can be confident in your belief that most people can learn it and that it's a worthwhile endeavor.

The Value of Programming. There are at least a couple of good reasons for learning about data processing and programming. The first reason is that you're going to have to deal with computers in the future whether you like them or not. Practically every monetary transaction that takes place on a daily basis involves computers -- bank accounts, checking accounts, invoices from power and gas companies, and checkout lines at the grocery store, just to mention a few. If you haven't already been exposed to "errors" in one of these transactions, sooner or later you will be. Once you learn how a computer "thinks" and how it processes information, you'll perhaps be more tolerant of the mistake and, more importantly, you'll know who to contact and what information they need to correct the mistake.

The second reason is that, regardless of what your position is, you may soon find yourself having to make a decision about "buying" some data processing equipment or some programming. Suppose you're called upon by your job or your friends to set up a data processing system to keep track of a business or a civic association. Will a TI-99/4A get the job done? Do you need a printer, and what kind? Is 16K enough? Do you need one, two or three disk drives? You can't really answer these questions until you thoroughly understand the capabilities of each particular piece of equipment involved. And that's just the hardware, what about the software (programs)?

Even if you're willing and able to hire someone to write the programs for you, you're still going to have to select them. If you gave six or seven programmers a set of specifications on a business package to keep track of inventory, you could get quotes ranging from $200 or $300 dollars up to perhaps $3000. Which one is right? Unless you know something about programming, there's no way for you to know who is giving you the best deal. Your specifications may not have been clear because you didn't understand what it was that they needed. One person's interpretation of an inventory program may not be the same

as another's. Actually, they may be quoting on the same thing, except the lower priced programmers may already have programs which simply need to be "modified" for your use. If their original programs are good, modifications may only take a couple of hours, while the higher priced programmer may have to spend 50-60 hours just to get started. The point we're trying to make is that a little bit of understanding regarding what goes into data processing will make you a better "buyer" of hardware and software.

Basic Logic. You may have noticed by now that just about every time you turn around, you're being encouraged to expand your system. Some will tell you that LOGO is the only answer for learning to program; that a printer is a necessity for meaningful records; that a Speech Synthesizer is required for educational programs for children; or that Extended Basic and Mini Memory are required for serious programming. We're not saying that there isn't some truth to these statements; but how are you, the new user, to decide when it's time to expand and what you need. The answer to this question is that you must first understand what it is you have available before you can decide exactly where it falls short of your needs. You can then select expansions to your system based on those needs. The thought process and the logical considerations can be learned as well on Console Basic as they can be with a $5,000.00 system.

A prime example is the attached "Checkbook" program. We've cautioned you not to rely on that data as your sole source of record keeping unless you have a printer and the ability to make "hard copy" reports showing every transaction. Does this mean that the program is useless or that you need to go right out and buy a printer? The answer to both questions is "NO". You may get a printer and find out six months later that keeping up with the "posting" of the checks on a weekly basis is more bother than you think it's worth. On the other hand, you may decide that the information is worthwhile and you want to expand your system to include disk drive. With disk drive you'll be able to keep perhaps a years worth of checks all together and you'll be able to access any one of them at will. If that's what you decide, you will not have lost time by building your files on cassette, in fact you'll be way ahead of the game. All you need to do is write a small utility program that gathers all of the data from the individual (71 record) cassettes and combines it into one data file on disk. In most cases, you can make a few modifications to your cassette based programs to take advantage of the new capabilities. At this point, at least you're sure that you want it and that you need it.

Hopefully, you're as convinced now as you ever were that you want to do some programming -- so let's get started with the fundamentals.

Entering-Saving -Loading. Prior to going to a lot of time and effort to enter programs, be sure that you know the procedures for SAVEing and loading programs from cassettes and that your recorder and cassette cable hookup is working properly. Refer to your User's Reference Guide (URG) for instructions on "CONNECTING THE RECORDER". If you don't have an approved model or specially designed TI recorder, bear these things in mind:

1. Don't use batteries - they're not constant enough.
2. Don't use a stereo recorder. You may be able to save and load your own programs, but more than likely it won't be compatible with any others.
3. At a minimum, you should have a jack for "Ear" or "Monitor", "Remote", and "Microphone" or "Record".
4. Usually the Red pigtail goes in "Mic", the White in "Monitor", and the Black in "Remote".
5. It it's not new, clean the head and pinch rollers before you try it. Do this every four or five hours of operation. The rollers collect a red oxide coating from the cassettes. Even a slight build up can cause slippage which results in distorted sound and "bad data".

Now that you're ready, turn on the monitor and computer. After being greeted by the "Home Computer" display, hit any key, as instructed. Next, enter "1" for TI Basic. When the "TI BASIC READY", with prompt (>) and blinking block (cursor) appears, you're ready to begin inputting programs. Enter 10 or 15 lines from one of the programs in this manual or a sample program from your owner's manual. Make sure you hit the "ENTER" key after the last line you put in. To test the recorder, following the prompt (>), type SAVE CS1, all in CAPITAL letters, and hit the "ENTER" key. The screen will now show:

```
>SAVE CS1

* REWIND CASSETTE TAPE          CS1
            THEN PRESS ENTER
```

From this point on, simply follow the instructions on the screen. Complete details of each statement are found in the "User's Reference Guide". When checking your data, if you get the error message "ERROR - NO DATA FOUND", it probably means the volume setting is too low. Increase it slightly, then hit the "C" to check it again. If the error "ERROR IN DATA DETECTED" occurs, the volume is probably too high. Reduce volume slightly, hit the "C" and check it again. Continue this process until you are successful. When you have found the proper setting, make a note of the settings or scotch tape the volume and tone settings so that they cannot be changed by accident in the future. Now you may begin inputting programs.

Most of the programs presented in the early part of this manual are broken down into a number of subroutines (GOSUBs) to make them easier to understand and easier to enter and check (debug). The general sequence of each program is usually found within the first 10-20 lines. You may use the NUMBER command or enter line numbers as you go. Although REMarks are not necessary to the running of the program, we suggest that you enter these completely and keep all line numbers the same, so that you have a written copy of exactly what is in the computer.

Every effort has been made to make all of our program listings and example programs appear just as they will on your screen (i.e. 28 character lines). When you key in a line that "rolls over" to the second or third line, the characters on your screen should appear the same as the line listing. To avoid problems, make sure you always pick up the number following the word THEN in an IF-THEN statement. Sometimes this will appear as the first number on the next line.

Also, if you type in a REM statement with two spaces following the REM (as our line listing will show), when you LIST it, it will appear with 3 spaces following. Every time you EDIT a REM statement, it will add another space. To be consistent with our listing, and to avoid confusion, enter just one space following a REM and then key in the actual remark.

Computers are very touchy about electrical shock, interference, and/or static electricity. If you're running a dishwasher, washing machine, or other major appliance while you're entering data, a surge or drop in electrical current could cause you to lose the information that you've just put in. For that reason, we suggest that you SAVE the program after each 40 or 50 lines to prevent you from having to start all over.

After the program is entered, and you return to the prompt, type RUN. Unless you're an extremely skilled typist, it's likely that the program will stop at some point and give you an error message with a line number reference. Follow the instructions under LIST in the URG to view 4 or 5 lines above and below the referenced number. Compare this with the printed copy and add lines that have been missed or make changes as required by either retyping the incorrect line or using the EDIT command. If the line contains VARIABLES (such as X, A, C$, etc.) the error may not really be in that line, but in the line that creates those variables. Look through your line listing until you find the line that controls the variable and make sure it's entered properly. More detailed information on debugging can be found in Chapter 3. Continue this process until the program runs properly and SAVE as instructed above.

Programs previously SAVEd on your recorder, and successfully checked, at a given volume and tone setting, should load again without any problem. Programs recorded on another recorder, or programs purchased on cassette, may require some tone and volume adjustments as mentioned above. You load an existing program by bringing the system to a prompt and then typing: OLD CS1. The screen will then give you instructions on completing the process.

Cassettes. The above instructions assume that all programs are recorded at the beginning of each tape cassette, and that no more than one program is on each side of the cassette. Special 5, 10, and 15 minute cassettes are available at computer stores and some other retail stores, designed specifically for this application. These are preferable to the longer 30 or 60 minute cassettes. It's possible to use the longer cassettes to SAVE many programs on one tape if you keep close track of the exact setting on the digital counter; however, should you lose the data or have a leader tear lose, all of the programs will have to be recreated instead of just one. If you're going to enter five or six programs, you're probably going to need a dozen or more cassettes. We suggest that you always make at least two copies of every program before you shut off your system. Cassettes can wear out, leaders tear loose, and magnetic objects can destroy data. If you don't have a printer, that cassette is the only record of your program - you don't want to lose it.

The Manuals. Take care of your manuals! Unlike the instruction manual you get with a television or refrigerator, which you read once and then file away, the manuals that go with your computer will be your constant companions and guides for at least the first six months to a year. TI's "Beginner's Basic" and "User's Reference Guide" are extremely well done and do a fantastic job of teaching the individual commands available on the TI-99/4A. Unfortunately, even the best written manuals can often times be confusing to the new user. Manufacturers, such as TI, must serve the entire market; therefore, they have to point out all of the possible uses and capabilities of their particular system. This means they have to get into the area of algorithms, arctangents, cosines, disk drives, file structures, etc. Many of these things will be of interest to only a small segment of the users such as engineers, scientists, and advance programmers. Since our concern is only with Console Basic, with no peripherals (additional equipment), and the creation of primarily "friendly" user written programs, a lot of this information need not be studied. We're going to make a quick review of these two manuals pointing out the most important aspects and certain relationships that exist between commands; however, before we do, we'd like you to bear in mind two important points. First, don't try to memorize the information presented. In other words, remember that there is a command that will generate a random number, but don't necessarily try to remember exactly how to code it in. This will save a great deal of time and speed up your learning process. Many of these command, while they are occasionally necessary, are used infrequently. For these commands, even if you memorize it today, the lack of use will cause you to forget it by the next time you need it. For the more important commands, you'll learn them after you've typed in several complete programs. No deliberate attempt at memorization will be required. Second, and TI points this out in several places, you need to EXPERIMENT. The error statement is the programmer's best friend. In our finished programs we don't want to see error statements; but, while programming, it's through the error statement that we find out what's possible and what's going on in the program. If you think you have an idea that might work, by all means try it. The computer will not be damaged, nor will it damage the recorder or other device hooked to the computer. With these words of advice, let's move on to the "Beginner's Basic" Manual.

"Beginner's Basic". This manual is a great aid for those who have had no programming experience. If you fall into this category, you are well advised to type in all of the examples given. Everything presented will be used over and over again in day-to-day programming, with two exceptions. First, the LET statement is totally unnecessary. They state that it is optional, but in actual practice it just isn't used. If you want to create a value for a variable or string simply use the short version:

```
>10  A$="THIS IS A STRING"
>20  B=32*J
```

Second, the immediate mode, except for loading and saving information, will primarily be used in the future by the programmer for finding errors, and testing ideas, rather than by the actual user of a prepared program.

By the time you've completed this book you should have a good understanding of the line numbering concept and the way the computer moves from one statement to the next, and the idea of letters and even numbers being represented by other numbers (ASC Codes). You should also understand the major commands such as CALL CLEAR, PRINT, INPUT, GOTO, GOSUB, and the FOR - NEXT loop. Don't worry as much about the Graphics and Sound capabilities if you don't understand them at first. They'll become clearer as we begin to work with actual programs. You might want to paperclip or tab the pages with the ASC codes, the shorthand graphics code, and the color codes, as you'll refer to these often.

"User's Reference Guide". After you've begun programming, this is the manual that you'll refer to more often, since it goes into much greater detail for each command.

Read the entire "General Information" section paying particular attention to the "Cassette Interface Cable" instructions. Don't be confused by the instructions for loading data from a cassette. The LOAD DATA command is available on separate modules only. If you're loading from a cassette which you have SAVEd you'll use the OLD CS1 command.

Following this section there's a section called "General Information". If you've worked through "Beginner's Basic" you'll already understand most of this. Pay particular attention to the "Special Keys". Get accustomed to using these function keys. They have a slightly different use while in TI Basic, EDIT or NUMber mode. Generally, they permit you to make corrections without having to retype entire lines. For the time being you can disregard the information on "Numeric Constants". If you needed this information for your programming you would probably already understand it, otherwise you'll probably never use it.

You'll need to be aware of all of the commands in the section entitled "Commands", and the "General Program Statements". The basic commands are what you'll use while programming, running, saving, loading, and debugging your program. Most of them are not and cannot be used as lines of a program. They're used in the immediate mode only. The "General Program Statements" are mostly repeated from the "Beginner's Basic" and should already be well in mind. The biggest confusion in the "Input-Output Statements" section for beginners seems to be the RESTORE, DATA, and READ statements, and the use of the PRINT command, particularly with regard to numbers. We're going to cover this later in this chapter so we won't go into detail here.

The "Color Graphics" and "Sound" sections again simply repeat much of what was in "Beginner's Basic". Graphics and sound are not extremely difficult, but coding characters can be time consuming. You'll find many examples of color and sound uses in the Building Blocks and Tank Attack programs which are found at the end of Chapter 2.

The next two sections in the URG, "Built-in Numeric Functions", and "Built-In String Functions" can look rather scary to the beginner. Most of the numeric functions, with the exception of RANDOMIZE and RND, while necessary for scientists, engineers, and serious mathematicians, are seldom

used by the average enthusiast. Be aware of the difference between RND and RANDOMIZE. The string functions however, are extremely important. These are used time and time again to convert numbers to strings, strings to numbers, and either to ASC character codes, etc. You'll find almost every one of them used in the two subroutines at the end of this chapter -- one for screen placement of messages and the other for right justification of numeric data. The user-defined Function called DEF can be handy where a single calculation is needed repeatedly throughout a program and can sometimes be used in place of complete subroutines.

Arrays are a study in themselves and we will devote an entire chapter to their use. In addition to being useful, they're interesting to work with. The 16K memory of Console Basic makes the two and three dimensional numeric arrays less valuable than they might otherwise be on a larger system, but single dimensional arrays and multi dimensional string arrays will be used frequently when we get into file handling and sorting.

Skipping over the GOSUB for a moment, let's get straight to the file processing. Much of the information presented in the URG pertains to disk drives and not to cassette recorders. In particular, look at the section entitled "Cassette Recorder Information". Now, put a clean tape in your recorder and enter the following example. It'll show you the essentials you need to get started.

```
>100 OPEN #1:"CS1",DISPLAY, O
 UTPUT,FIXED
>110 X$="THIS IS A TEST"
>120 FOR I=1 TO 5
>130 PRINT #1:X$&STR$(I)
>140 NEXT I
>150 CLOSE #1
>160 OPEN #1:"CS1",DISPLAY ,I
 NPUT ,FIXED
>170 FOR I=1 TO 5
>180 INPUT #1:X$
>190 PRINT X$
>200 NEXT I
>210 CLOSE #1
>220 END
```

This program opens a file; creates a string called X$; changes the value of I to a string and adds it to X$; prints X$ to the cassette recorder 5 times; closes the file; opens the file again; inputs the string called X$; prints it to the screen; and then closes the file a second time. There is a lot more to files than this, but this may give you a place to start. Trying changing what you print to the file and how you print it when it returns.

To complete the review of the manual we need to discuss the GOSUB. If you want to learn to program and design programs the easy way, this section is worth reading over and over again. Look at the two programs included in Chapter 2 and imagine what they might look like without GOSUBs. Later in this chapter we're going to review how to right justify numeric data and how to print messages to the screen. Both of these are treated as subroutines and many more examples are offered throughout the manual. There are times when some of these subroutines can be eliminated and programs will actually run faster by using GOTO statements instead; but the ease in preparation and understanding for the beginner cannot be overstated. Following are two useful GOSUBs to get you started.

GOSUBs. As you'll see in the coming chapters, programs can easily be built one section at a time. Each individual section of the program is like a miniature program in itself, performing only one task. To get started with programming we've included in this chapter two commonly used subroutines -- one for right justifying numbers and the other for printing messages to any point on the screen. In order to make these subroutines work, we've used: DATA, READ, and RESTORE statements; several FOR-NEXT loops; and most of the string handling statements.

DATA Statements. The following base program will be used to demonstrate how DATA, READ and RESTORE statements work together to "feed" information to a program. After you've entered this program, we'll discuss it and add other statements and subroutines in place of the REMark statements to demonstrate number and message placement techniques.

```
>100 REM
>110 CALL CLEAR
>120 DATA 1025.86,-329,1.98,.
 22
>130 DATA THIS PROGRAM READS
 NUMBERS,FROM LINE 120 AND SE
 NTENCES,FROM LINE 130 USING
 THE READ,STATEMENT,"",""
>140 RESTORE 130
>150 FOR I=1 TO 6
>160 READ A$
>170 PRINT A$
>180 NEXT I
>190 RESTORE 120
>200 FOR I=1 TO 4
>210 READ AMT
>220 REM
>230 PRINT TAB(5);AMT
>240 TOTAL=TOTAL+AMT
>250 NEXT I
>260 REM
>270 PRINT TAB(5);TOTAL
>280 REM
>290 GOTO 290
```

Running this program clears the screen and then produces the following display:

```
THIS PROGRAM READS NUMBERS
FROM LINE 120 AND SENTENCES
FROM LINE 130 USING THE READ
STATEMENT

     1025.86
    -329
     1.98
     .22
     699.06
```

When this program stops it is "idling" in line 290, continuously sending itself back to the same number. This prevents the program from stopping and giving the **DONE** message. To understand the DATA, READ and RESTORE statements, do a FCTN-4 to interrupt (BREAK) the program, enter the following lines which will replace lines 100 and 280 above, and then RUN the program.

```
>100 TRACE
>280 UNTRACE
```

Now you see the same program, except each sentence or number is preceded by a series of numbers enclosed in brackets. Using the TRACE command we have shown you the sequence of events. Lines <100> and <110> do not appear because they were CLEARed off the screen when the program went through line 110. The program runs directly through all of the lines from 100 to 170 before printing anything to the screen. As the program went through lines 120 and 130, it stored in memory the DATA contained in those two lines

and the line number in which the DATA was located. When it went through line 140 it set a "pointer" in its memory to the first data statement in line 130. When it hit the READ statement in line 160 it assigned that first element of data (i.e. the words "THIS PROGRAM READS NUMBERS") to the variable A$, and moved its "pointer" to the second element. In line 170 it then printed that to the screen. Notice that after it is printed, the next series of numbers is <180><160><170>. After the NEXT statement in line 180, the computer does not go back to the DATA statement, it goes to the READ statement in 160. To find the DATA it takes the next element of data (the second one in line 130) based on the position of the "pointer" in its memory. This continues until all words are read and the program resets its pointer in line 190. It is now looking at the numeric data which was stored when it went through line 120. Each time through the FOR-NEXT loop from 200-250, the program prints a number to the screen at TAB(5) and then adds that number to a variable named TOTAL. At the completion of the loop, the value of TOTAL is also printed to the screen at TAB(5). Do a FCTN-4, enter the following, and then RUN the program again.

```
>100 REM
>190 REM
```

Running this program will print the sentences correctly; however it will error out in line 210 when it attempts to read the AMT. Since we removed the RESTORE statement from line 190, the "pointer" in memory had no more data to read. It had used all of the six elements from line 120 and had not been positioned to any other point. Add back the RESTORE 120 statement to line 190 and we're going to use this same program to demonstrate how to print numbers in columns.

Number Format. You'll notice in this program that we've printed four numbers and a total to the screen, all at TAB(5), yet the numbers do not line up as we normally like to see a column of numbers. For a normal column of numbers, we would like all of the decimal points to line up and we would want ".00" after whole numbers. Leave the program in the computer and build a subroutine beginning in line 1000 as follows:

```
>1000 REM DECIMALS & SPACE
>1010 L=LEN(STR$(AMT))
>1020 AMT$=STR$(AMT)
>1030 FOR J=1 TO L
>1040 IF SEG$(AMT$,J,1)="." T
 HEN 1060
>1050 NEXT J
>1060 REM
>1070 ON L-J+2 GOTO 1080,1100
 ,1100,1120
>1080 AMT$=AMT$&".00"
>1090 GOTO 1130
>1100 AMT$=AMT$&".0"
>1110 GOTO 1130
>1120 AMT$=AMT$
>1130 IF LEN(AMT$)=10 THEN 11
 60
>1140 AMT$=" "&AMT$
>1150 GOTO 1130
>1160 RETURN
```

Replace or add following:

```
>220 GOSUB 1000
>230 PRINT TAB(5);AMT$
>260 AMT=TOTAL
>265 GOSUB 1000
>270 PRINT TAB(5);AMT$
```

Running this program converts all numbers to a string called AMT$ before it is printed to the screen. In lines

1010-1060 we determine the length of the number after it is converted to a string, and we find the position of the decimal point, if there is one. Based on the position of the decimal in relation to the length, we add a ".00", "0", or nothing. After we've added the required ending, we check the length to see if it equals 10. If not, we keep adding spaces and checking again until it does. When all numbers are structured properly we do a RETURN and allow the program to print the finished AMT$ to the screen.

There is a more thorough treatment of this problem later in this manual and some shorter methods of accomplishing the same thing. This routine will work with most numbers, provided they don't exceed two places after the decimal.

Screen Placement. Instead of scrolling information to the screen, it is possible to print directly to a specific row and column using a simple screen placement subroutine. This is not as fast as scrolling; however, when you get into graphics and game programs, scrolling isn't always possible.

```
>2000 REM SCREEN PLACEMENT
>2010 FOR I=1 TO LEN(MSG$)
>2020 CALL HCHAR(3,4+I,ASC(SE
 G$(MSG$,I,1)))
>2030 NEXT I
>2040 RETURN
```

Add the following:

```
>285 MSG$="THAT'S ALL FOLKS .
  . ."
>287 GOSUB 2000
```

This is really a very simply subroutine and the basic structure is used time and again in the programs included in this manual. A message subroutine needs three items of information in order to perform its function. It needs to know the message to be printed, which we usually set up as MSG$; it needs the row on which it is to be printed; and it needs a starting column. Using the HCHAR command and a FOR-NEXT loop, we can evaluate each character of the message, turn it into its ASC number, and CALL that character to a specific point on the screen. The I value in the FOR-NEXT loop insures that each character is printed sequentially to the screen.

Building Programs. Notice how we started with a base program above and gradually made changes to it. The additional features we built into it were added as subroutines at the end, usually with a totally different numbering sequence. Occasionally small changes we're required in the base program to route it through the subroutine. Before any new subroutines were added we were sure we had a working program that had no errors. This is the basic philosophy of programming that this manual teaches and is the subject of our next chapter.

CHAPTER TWO

Programming Philosophy

GENERAL. Programming is not difficult — it's time consuming! If you can make yourself believe this statement, you're better than 80% on the way to becoming a programmer. This chapter could easily be called "Logical Thinking" or "Problem Solving", because the key to programming is more the state of mind than it is the use of any highly developed manual skills. You're not the one that needs to learn. Anything you want the computer to do, you already know how to do. It's the computer that needs educating. To borrow a phrase from a popular movie, "What we have here is a failure to communicate". Essentially, it's like trying to teach a three year old, with a limited vocabulary, how to perform a complex task. The 99/4A understands less than 100 words. The way to handle the problem is to break it down into very small steps, each of which can be explained with just a few words.

If you weren't a carpenter and someone gave you a box of tools and then dropped off three truck loads of bricks, lumber, shingles, etc., could you build a house. At first glance, most of us would probably say "No". But, you could probably drive a single nail, measure a board, cut a 2 X 4, or paint a wall. All you really need to get the job done are some detailed instructions. That's all a computer program is, and the programmer is the one that decides what those instructions will be. For those that aren't used to working with computers, the problem is they tend to think too fast. Their instructions on building the house would include statements like: "construct a foundation", "put up the four walls", "put on a roof". The job is still too big for the novice to understand. Let's put this in computer terms.

If you look at at a program listing that goes from 100-4500, a total of 441 lines, it's like looking at three loads of material. If you're thinking of a screen display that'll have one plane shooting down another, you're thinking about "putting up four walls". You need to get your thinking down to the "nail driving" stage. When you think about a program, you have to get your thinking down to the point where you're only concerned with moving one character or creating one variable. Fortunately, you don't have to start from "scratch". Most programs aren't that different from each other; in fact, there are only about four or five basic structures.

Types of Programs. The four basic types of programs that you'll generally be exposed to can be broken down into "Utility", "Functional", "Educational" and "Game" type programs. "Functional" programs usually involve the use of data files and can be further broken down into "Input/Update" type or "Call Out/Display" type programs. The other three types are normally self contained and do not require the use of data files for operation. For any given type of program, you'll find that the sequence of events is remarkably similar from one program to the other, only the details change.

What we're going to do in this chapter is give you the basic outline for each — all you have to do is fill in the details. Each of these examples is a complete program and you may even pick up some other useful ideas on PRINTing, TABing, and CALL KEY's as you enter them.

Utility Programs. Utility type programs are generally short (1ØØ-2ØØ Lines) and they're designed to serve a single purpose. The checkbook balancing programming in the back of the URG is an example. Another example might be a program to calculate the monthly payments on a home if amortized over 3Ø years at 12% interest. Think about things that you've had to spend time calculating in the last month such as: a carpenter who does calculations regarding "Board Feet"; a businessman who needs to calculate a rate of return; or an engineer who needs to figure arcs and angles. These are all possible applications for the home computer. To start a program like this, key in the following program to begin with:

```
>1ØØ CALL CLEAR
>11Ø REM DISPLAY INFORMATION
>12Ø GOSUB 1ØØØ
>13Ø REM INPUT INFORMATION
>14Ø GOSUB 2ØØØ
>15Ø REM PROCESS INFORMATION
>16Ø GOSUB 3ØØØ
>17Ø REM DISPLAY RESULTS
>18Ø GOSUB 4ØØØ
>19Ø GOTO 1ØØ
>1ØØØ REM INSTRUCTIONS
>1Ø1Ø PRINT "THIS IS WHERE YO
 U GIVE IN-  STRUCTIONS TO TH
 E USER."::
>1Ø2Ø PRINT "THIS PROGRAM WIL
 L ACCEPT    TWO NUMBERS AND
 PERFORM A   CALCULATION ON T
 HEM":::
>1Ø3Ø PRINT "HIT ANY KEY TO C
 ONTINUE . ."
>1Ø4Ø CALL KEY(3,KY,ST)
>1Ø5Ø IF ST=Ø THEN 1Ø4Ø
>1Ø6Ø RETURN
>2ØØØ REM INPUT SECTION
>2Ø1Ø CALL CLEAR
>2Ø2Ø INPUT "ENTER ANY NUMBER
 : ":A
>2Ø3Ø INPUT "ENTER ANOTHER:
 : ":B
>2Ø4Ø RETURN
>3ØØØ REM PROCESS INFO
>3Ø1Ø CALL CLEAR
>3Ø2Ø PRINT "THE PROGRAM HAS
 ACCEPTED    THE DATA AND IS
 NOW ADDING  THE NUMBERS"
>3Ø3Ø C=A+B
>3Ø4Ø FOR I=1 TO 5ØØ
>3Ø5Ø NEXT I
>3Ø6Ø RETURN
>4ØØØ REM DISPLAY RESULTS
>4Ø1Ø CALL CLEAR
>4Ø2Ø PRINT "FIRST NUMBER =
  ";A::
>4Ø3Ø PRINT "SECOND NUMBER=
  ";B
>4Ø4Ø PRINT TAB(17);"____"::
>4Ø5Ø PRINT "TOTAL         =
  ";C:::
>4Ø6Ø PRINT "HIT ANY KEY TO C
 ONTINUE. . ."
>4Ø7Ø CALL KEY(3,KY,ST)
>4Ø8Ø IF ST=Ø THEN 4Ø7Ø
>4Ø9Ø RETURN
>RUN
```

When you RUN this program the first thing you get is a display that simply tells you what the program will do and what kind of input it's going to be expecting. It's not uncommon to end this subroutine with a CALL KEY statement as we have done in line 1Ø4Ø. After you hit a key, the program goes on to the INPUT subroutine where you are asked to

enter two numbers. The program then processes the information, displays it on the screen, and again waits for you to hit any key.

In reality, you wouldn't write a program this long, complete with subroutines, to add A & B. If that's all you needed to do, you wouldn't need the computer in the first place. Normally, a utility program has several input sections and several sections which perform different calculations. After you've given instructions to the user on what the program does, your first INPUT section (GOSUB 2000) might actually be a "Menu". This is a common term for a listing of options from which the user can select. The "Money Planner" program at the end of Chapter 11 has a menu in lines 370-500. If you need a menu, put this in 2000 and build separate INPUT subroutines beginning at 2200, 2400, 2600, etc. If you have different sections for calculations, start them at 3200, 3400, 3600, etc. You probably won't think of all of the subroutines necessary right at the beginning of a program. Consider the main ones and spread out your subroutines. As the program develops, if you need another, just add it to that section of the program.

The next step is to begin developing the individual subroutines. Get in just the essential information required to make the program operational. Initial instructions can be rather sparse, in fact they may change by the time you complete the program. The spacing on the menu may not be exactly right. As long as it states the options and has an INPUT to accept a choice, that's all you need. If you're asking for a date or an amount, you may eventually want to test this input for validity, but you can skip a lot of this at this point. Use descriptive variables for your INPUTs like ANS for "Answer", AMT for Amount, DATE for "Date", etc. Think only about the one question you're working on. If you don't know how to perform a particular calculation, try breaking it down into 2 or 3 smaller statements. As you'll see in Chapter 10 (Condensing), there will be plenty of time later to go back and take out unnnecessary lines.

As you complete each subroutine, RUN your program and work out the "bugs". Any misspellings, bad values, etc., can be caught at this point. If you start with a running program, as we've shown above, you should be able to keep it in running condition throughout the development of the actual program. After each subroutine is running, SAVE your program on cassette before beginning the next section. DO NOT RESEQUENCE THIS PROGRAM.

Now that you have an idea of how we structure a program, let's go on to the format for the other three types of programs.

<u>Game Programs.</u> Ideas for games are easy to come by. All you have to do is think about the games that children and adults play such as: card games, board games, gambling games, baseball, football, etc. What isn't so easy is finding ways to create the activity on the screen. Often times our ideas are simply unachievable because of the limitations of the system. As far as the 99-4/A is concerned, the main thing to remember is that you can only move 1 character at a time. This means that you can't shift the entire screen at one time, so you can't very well have a moving road or a number of objects flying at you at the same

time. Two or three characters sitting next to each other can be erased and replaced at another point and you can get a pretty close approximation of multiple movement. The red and blue tanks in "Tank Attack" each consist of three characters. Beyond three characters, the character by character building process becomes obvious to the user.

Here's a good basic layout for a game program. Almost all of these sections will be required for just about any game.

```
>100 CALL CLEAR
>110 REM INITIAL VARIABLES
>120 GOSUB 1000
>130 REM OPENING DISPLAY
>140 GOSUB 2000
>150 REM MAIN GAME LOOP
>160 GOSUB 3000
>170 REM ACTION NO 1
>180 GOSUB 4000
>190 REM ACTION NO 2
>200 GOSUB 5000
>210 REM EXPLOSION
>220 GOSUB 6000
>230 REM SCORE ROUTINE
>240 GOSUB 7000
>250 REM PRINT ROUTINE
>260 GOSUB 8000
>270 GOTO 130
>1000 REM VARIABLES
>1090 RETURN
>2000 REM DISPLAY
>2090 RETURN
>3000 REM MAIN LOOP
>3090 RETURN
>4000 REM ACTION 1
>4090 RETURN
>5000 REM ACTION 2
>5090 RETURN
>6000 REM EXPLOSION
>6090 RETURN
>7000 REM SCORE
>7090 RETURN
>8000 REM PRINT
>8090 RETURN
```

This program is a running program, although you really won't see anything on the screen because we haven't put anything in the subroutines. Following are some additions that we made to this program. Add these directly to the above and RUN it again. This program defines a couple of characters, builds a quick display and prints two colored blocks to the screen. If you hit the period, you'll get a "tone" indicating an "explosion". Hit the "X" and the program will start over. We've put the statements into the print subroutine to accept a message but it isn't in use yet.

```
>165 IF KY=88 THEN 130
>1010 CALL CLEAR
>1020 CALL CHAR(128,"FFFFFFFF
 FFFFFFFF")
>1030 CALL CHAR(136,"FFFFFFFF
 FFFFFFFF")
>1040 CALL COLOR(13,7,1)
>1050 CALL COLOR(14,3,1)
>1060 CALL SCREEN(12)
>2010 CALL CLEAR
>2020 CALL HCHAR(3,3,128,28)
>2030 CALL HCHAR(21,3,128,28)
>2040 CALL VCHAR(4,3,128,17)
>2050 CALL VCHAR(4,30,128,17)
>3010 REM MOVE #1
>3020 GOSUB 4000
>3030 REM MOVE #2
>3040 GOSUB 5000
>3050 CALL KEY(3,KY,ST)
>3060 IF ST=0 THEN 3050
>3070 IF KY=88 THEN 3100
>3080 IF KY<>46 THEN 3050
>3090 GOSUB 6000
>3095 GOTO 3010
>3100 RETURN
>4010 R1=INT(15*RND)+5
>4020 C1=INT(20*RND)+5
>4030 CALL HCHAR(R1,C1,128)
>5010 R2=INT(15*RND)+5
>5020 C2=INT(20*RND)+5
>5030 CALL HCHAR(R2,C2,136)
```

```
>6010 CALL SOUND(100,110,0)
>7010 SCR=SCR+50
>8010 FOR I=1 TO LEN(MSG$)
>8020 CALL HCHAR(R,C,ASC(SEG$
 (MSG$,I,1)))
>8030 NEXT I
```

To develop a complete game, just keep adding more individual statements; dress up the characters; get fancy with the sounds; etc.

The main difference in a game program is where you start programming. In a game program, always start with the most difficult and questionable portion of the program. Usually this involves some sort of movement. Perhaps you want to bounce something off the bottom of the screen to the top. First define a character in GOSUB 1000 as "FFFFFFFFFFFFFFFF". This is a solid character. Now begin in GOSUB 4000 and try to write a routine that will move this block the way you have in mind. Eventually you may want to recode this character to be a plane, monster, etc., but if you can't get the movement there's no sense in spending the time coding characters. When you're sure that what you have in mind will work, then begin working on the other "Action Routines". Lines below 1000 will generally control the movement through the various GOSUBs. The exception is GOSUB 3000. This routine controls most of the action of the game and may call on GOSUB 4000, 5000, 6000, and 7000. Leave the coding for the opening display until last, since you may run close on memory. If you run out of memory you can always get by without a "classy" opening display. Bear in mind that your routines may not be exactly as shown above. The "explosion" routine may be a "sinking ship" routine, or "Action No 1" might be a "Falling Rock", etc. Create them as required and write down the GOSUB number and what it does on a piece of paper next to you.

<u>Educational Programs.</u> Sources for educational programs are numerous. Children's workbooks are probably the best source. The inspiration for the "Building Block" program came from a subscription type book containing games and activities for children. It had a page with some triangles, circles and squares on it and the child was to cut them out and paste them on a piece of paper to make a design. The general layout for an educational type is sort of a cross between a utility type and a game. Following is the layout:

```
>100 CALL CLEAR
>110 REM INITIAL VARIABLES
>120 GOSUB 1000
>130 REM TEACHER INSTRUCTIONS
>140 GOSUB 2000
>150 REM OPTIONS
>160 GOSUB 3000
>170 REM SCREEN DISPLAY
>180 GOSUB 4000
>190 REM INPUT RESPONSE
>200 GOSUB 5000
>210 REM REWARD
>220 GOSUB 6000
>230 REM PUNISHMENT
>240 GOSUB 7000
>250 REM SCORE
>260 GOSUB 8000
>270 REM PRINT ROUTINE
>280 GOSUB 9000
>1000 REM VARIABLES
>1090 RETURN
```

NOTE: Add REMarks and RETURN for 2000 through 9000

This kind of program starts out with instructions to the educator (teacher or parent) telling them what the program does and how to use it. After

hitting a key the program cycles to the options subroutine where the educator selects from a Menu (list of choices) what he wants the child to learn. In math, this may be the level of the multiplication table, such as "7's". In geography, it may be "States" or "Capitals of States". It may also include options such as "Sequential Order" or "Random Order". Depending on the options chosen, the program will then go to a screen display. If these are short routines, you may have more than one screen display available and you can add it at 45ØØ. After the screen display, the child is challenged in some way and must respond in some manner. Depending on his response, the program is sent to either REWARD or PUNISHMENT and, in either case, it is then sent through the SCORE subroutine. Often times in this type of program, there is an end to the questioning. Once the child has completed all of the questions the program would return back to OPTIONS. One of the OPTIONS should be to view the SCORING summary.

This is similar to the utility type program in that you are often times working with INPUT statements; however, to be exciting for children, we also need to include moving or at least colorful graphics as a REWARD. Start programming this type by working on the SCREEN DISPLAY and INPUT responses. You can always change or enhance your reward later.

<u>Functional Programs.</u> These are by far the most complex of the programs and they usually include the use of data files for storing information. One program is usually used to create and maintain a data file. This would be like a program to INPUT names, addresses, and telephone numbers. After the names are added the information is stored on cassette for later retrieval. Usually the same program that permits entry of new items can also be used to change existing items (such as when a person's address changes). Following is the normal sequence for an Input/Update type program.

```
>1ØØ CALL CLEAR
>11Ø REM MAIN MENU
>12Ø GOSUB 1ØØØ
>13Ø REM OPEN FILE AND INPUT
 EXISTING DATA
>14Ø GOSUB 2ØØØ
>15Ø REM ADD NEW DATA ITEMS
>16Ø GOSUB 3ØØØ
>17Ø REM DELETE ITEMS
>18Ø GOSUB 4ØØØ
>19Ø REM CHANGE ITEMS
>2ØØ GOSUB 5ØØØ
>3ØØ REM DISPLAY OPTION
>31Ø GOSUB 6ØØØ
>32Ø REM COMBINE DATA AND PRI
 NT TO DATA FILE
>33Ø GOSUB 7ØØØ
>34Ø GOTO 11Ø
>1ØØØ REM MAIN MENU
>1Ø9Ø RETURN
```

NOTE: Add REMarks and RETURN for 2ØØØ through 7ØØØ

Writing a functional program, involving data files, begins with deciding what information is needed and how it will be stored in the file. In the case of the checkbook program we knew we had to have a check number, who it was to, the date, the amount, and the account number. We then had to decide how many characters we would allow for each item and see how many complete records we could get into a line of data. When you use data files you'll almost always be using the maximum length data line (192 characters) going to and from the data cassette. In Chapter 6 on data files

we'll discuss the reasons for this further; however, for now just accept the fact that it is the most efficient way to handle information. If you're working with a single recorder and Console Basic only, the size of an individual data file will be limited to what you can bring up "in memory" at one time.

Once you know how many items you can get in one 192 character data line, the next thing you need to do is determine how many data lines you can bring into memory. With Console Basic, an approximate figure would be about 73 or 14,000 Bytes/192. If you figure you need to reserve half of your memory for the program then you can only have approximately 36 lines of data or, in the case of the check entries, 36 X 6 per data line, or 216 check records. Review the "Budget Maintenance" program and you'll see that we had to use some others for Budget and YTD figures so this number was cut down. There's a reason for going through this explanation of data lines when discussing functional files, because it's this calculation that determines whether you even have a feasible idea. We knew that in order to have a meaningful checkbook program we had to have at least a month's worth of checkbook entries and a reasonable number of expense categories. If our calculations indicated that we could only get 10-20 check entries or 5-10 accounts, we wouldn't have a program worth writing.

If you think you're within a reasonable range, then set up your subroutines as outlined above. Begin your work in subroutine 2000. This is the subroutine that gets your data. Continue through each of the others as previously explained.

Call Out Programs. Call out programs are usually pretty easy to construct and are really copied in large part from the Input/Update Program. They are used for creating graphs, printing to a printer (if you have one), sorting records, etc. The format is as follow:

```
>100 CALL CLEAR
>110 REM MAIN MENU
>120 GOSUB 1000
>130 REM OPEN FILE AND INPUT
 EXISTING DATA
>140 GOSUB 2000
>150 REM SORT ROUTINE 1
>160 GOSUB 3000
>170 REM SORT ROUTINE 2
>180 GOSUB 4000
>190 REM DISPLAY 1
>200 GOSUB 5000
>300 REM DISPLAY 2
>310 GOSUB 6000
>320 GOTO 110
>1000 REM MAIN MENU
>1090 RETURN
```

NOTE: Add REMarks and RETURN for
2000 through 6000

Be consistent when writing one of these programs and use the same variables that you use in the Input/Update programs. This makes debugging much easier if you have problems. The Budget/YTD Display program is a good example of a straight "Call Out" Program.

Throughout this chapter we've talked about writing individual subroutines and running your program as you build it. Unfortunately, sometimes you're going to get error messages. The next chapter will hopefully give you a better understanding of why you got the message in the first place; how you can determine what the "real" problem is; and how to solve it.

```
********************
*    TANK ATTACK   *
*     V-PA131KJ    *
*    BY T CASTLE   *
********************
```

DESCRIPTION. Tank Attack is a single player game, designed for use with either joystick or keyboard, where player attempts to attain the highest score by shooting the computer controlled tank. Player is initially given three blue tanks. One is operative and moves up or down (using the down and up arrows on the keyboard or the joystick) in a column five positions to the right of center screen. Two additional tanks, in reserve, are shown on the right side of the screen. The computer controlled tank is positioned 6 columns to the left of center screen and moves randomly approximately 3 or 4 rows above, below, or directly in front of the blue tank. If the two tanks arrive "on line" with each other, either through player movement or random movement of the red tank, a varying amount of time is allowed before the red tank "shoots" the blue tank. The player may either move out of the way or "fire" using the joystick button or "left arrow" key on the keyboard. If the player fires before the red tank, a "bullet" is sent across the screen and an explosion sound and display is created.

Scoring is progressive, with the amount added for the first hit increasing at 2Ø, 4Ø, 6Ø, 8Ø, and 1ØØ thousand points. In the first round, the first hit is worth 5Ø points. By the last round, the first hit is worth 5ØØ points. After the first hit at each level, the amount added is double the normal amount for successive hits up to 1ØØØ points per "kill". The high score for each playing series is displayed at the upper left side of the screen. The current score is displayed and changed after each hit in the upper right portion of the screen. One additional tank is awarded at each 1Ø,ØØØ point interval. Below 2Ø,ØØØ points a number appears in the lower left portion of the screen which counts down to zero to aid the player in determining whether he can get on line and fire before being hit. If he arrives on line when the number is zero, he will be shot and lose his player. From 2Ø,ØØØ to 5Ø,ØØØ only the beginning number is displayed and no count down is provided. After 5Ø,ØØØ points, no number is displayed and the player must instinctively determine whether he has time or not.

NOTES. This program is layed out very similar to the game layout provided in Chapter 2. No attempt has been made to consolidate the lines. If you want to practice some line reduction techniques, try rewriting the scoring subroutine in line 237Ø-31ØØ using the ON __ GOTO or ON __ GOSUB command. Most of the initial subroutines set up to guide us through development are still located in lines 16Ø-37Ø.

To make this game more challenging, you might write an additional subroutine that randomly places a "shield" in the row just ahead of the blue tank. Change the random movement of the red tank so that it "homes" in on the blue tank. The only place to hide would be behind the shield.

```
100 REM  ***************
110 REM  * TANK ATTACK *
120 REM  ***************
130 REM  BY T CASTLE
140 REM   AMLIST V-PA132KJ
150 REM
160 CALL CLEAR
170 REM  SET INITIAL VALUES
180 GOSUB 540
190 REM  DETERMINE JS OR KB
200 GOSUB 900
210 REM  SCREEN DISPLAY
220 REM  AND CALL KEY
230 GOSUB 3320
240 REM  BEGIN GAME
250 GOSUB 4030
260 REM  DISPLAY EXTRA TANKS
270 GOSUB 1090
280 REM  PRINT BLUE TANK
290 GOSUB 2330
300 REM  PRINT RED TANK
310 GOSUB 1330
320 REM  MOVE RED TANK
330 GOSUB 1380
340 REM  CHECK FOR "REDO"
350 CALL KEY(3,KEY,KBSTAT)
360 IF KBSTAT=0 THEN 410
370 IF KEY=6 THEN 230
380 REM  MOVE BLUE TANK
390 REM  & CHECK FIRE STATUS
400 REM  FOR BOTH TANKS
410 GOSUB 1470
420 IF L=1 THEN 440
430 IF L1=1 THEN 500 ELSE 31
0
440 L=0
450 REM  COUNTS TANKS LOST
460 ST=ST+1
470 GOSUB 1090
480 Q1=0
490 IF ST=TT THEN 230 ELSE 2
90
500 L1=0
510 REM  SCORES HIT TANKS
520 GOSUB 2380
530 GOTO 310
540 CALL SCREEN(12)
550 REM  DEF BLUE TANK
560 DATA 128,3F010FFFFF3F1F0
F
570 DATA 129,F0F8FFFFFFFFFFF
F
580 DATA 130,0000E0FCFCF8F0E
0
590 REM  DEF RED TANK
600 DATA 136,0000073F3F1F0F0
7
610 DATA 137,0F1FFFFFFFFFFFF
F
620 DATA 138,FC80F0FFFFFCF8F
0
630 REM  DEF FULL,BLANK&SHOT
640 DATA 132,FFFFFFFFFFFFFFF
F
650 DATA 139,000000000000000
0
660 DATA 140,3C3C00000000000
0
670 REM  DEF EXPLOSION-LTR N
680 DATA 131,006666001828414
1
690 DATA 141,006666001828414
1
700 DATA 133,F0F0F0F0F0F0F0F
0
710 DATA 134,0F0F0F0F0F0F0F0
F
720 REM  DEF COLOR SETS
730 DATA 3,2,1,4,2,1,13,5,1,
14,9,1
740 FOR I=1 TO 13
750 READ A,A$
760 CALL CHAR(A,A$)
770 NEXT I
780 FOR I=1 TO 4
790 READ A,B,C
800 CALL COLOR(A,B,C)
810 NEXT I
820 REM  START POS BLUE
830 R1=10
840 C1=21
850 REM  START POS RED
860 R2=12
870 C2=10
880 RETURN
890 REM  CHOOSE JOY OR KEYBD
900 CALL CLEAR
910 PL1=5
920 PL2=8
```

```
930 MSG$="ENTER 1 OR 2"
940 GOSUB 4170
950 PL1=7
960 PL2=8
970 MSG$="1. KEYBOARD"
980 GOSUB 4170
990 PL1=9
1000 PL2=8
1010 MSG$="2. JOYSTICK"
1020 GOSUB 4170
1030 CALL KEY(5,KBA,KJSTAT)
1040 IF KJSTAT=0 THEN 1030
1050 IF KBA=50 THEN 1070
1060 IF KBA=49 THEN 1070 ELS
E 1030
1070 RETURN
1080 REM  CALCS EXTRA TANKS
1090 EX1=TT-1-ST
1100 EX2=TT-10-ST
1110 IF EX1>=9 THEN 1220
1120 EX3=EX1*2+2
1130 FOR I=4 TO EX3 STEP 2
1140 CALL HCHAR(I,28,128)
1150 CALL HCHAR(I,29,129)
1160 CALL HCHAR(I,30,130)
1170 NEXT I
1180 FOR I=EX3+2 TO 22 STEP
2
1190 CALL HCHAR(I,28,32,3)
1200 NEXT I
1210 GOTO 1280
1220 MSG$=STR$(EX2)
1230 PL1=2
1240 PL2=29
1250 GOSUB 4170
1260 EX1=9
1270 GOTO 1120
1280 EX1=0
1290 EX2=0
1300 EX3=0
1310 RETURN
1320 REM  PRINT RED TANK
1330 CALL HCHAR(R2,C2,136)
1340 CALL HCHAR(R2,C2+1,137)
1350 CALL HCHAR(R2,C2+2,138)
1360 RETURN
1370 REM  MOVE RED TANK
1380 RANDOMIZE
1390 B=INT(((R1-SK)-(R1+SK)+
1)*RND)+(R1+SK)
1400 IF B=R2 THEN 1390
1410 IF B>=24 THEN 1390
1420 IF B<=4 THEN 1390
1430 CALL HCHAR(R2,C2,139,3)
1440 R2=B
1450 GOSUB 1330
1460 RETURN
1470 RANDOMIZE
1480 J=INT((J1-J2+1)*RND)+J2
1490 FOR IC=1 TO K
1500 IF L=1 THEN 1730
1510 IF J-IC>=0 THEN 1520 EL
SE 1600
1520 IF LV<3 THEN 1560
1530 IF LV<6 THEN 1540 ELSE
1600
1540 MSG$=STR$(J)
1550 GOTO 1570
1560 MSG$=STR$(J-IC)
1570 PL1=23
1580 PL2=3
1590 GOSUB 4170
1600 GOSUB 1780
1610 IF L1=1 THEN 1730
1620 IF IC>=J THEN 1650
1630 GOSUB 2010
1640 GOTO 1750
1650 IF R1=R2 THEN 1660 ELSE
 1750
1660 CALL SOUND(1000,-3,2)
1670 FOR F=C2+4 TO C1
1680 CALL HCHAR(R2,F,140)
1690 CALL HCHAR(R2,F,139)
1700 NEXT F
1710 GOSUB 3110
1720 L=1
1730 IC=K
1740 CALL HCHAR(23,3,32,2)
1750 NEXT IC
1760 RETURN
1770 REM   MOVES BLUE
1780 IF KBA=49 THEN 1790 ELS
E 1850
1790 CALL KEY(5,KEY1,STAT1)
1800 IF STAT1=0 THEN 1970
1810 IF KEY1=69 THEN 1880
1820 IF KEY1=101 THEN 1880
1830 IF KEY1=120 THEN 1900
1840 IF KEY1=88 THEN 1900 EL
SE 1970
```

```
1850 CALL JOYST(1,X,Y)
1860 IF Y=0 THEN 1970
1870 IF Y=4 THEN 1880 ELSE 1
900
1880 Y1=R1-1
1890 GOTO 1910
1900 Y1=R1+1
1910 IF Y1>23 THEN 1930
1920 IF Y1<4 THEN 1950 ELSE
1980
1930 Y1=23
1940 GOTO 1980
1950 Y1=4
1960 GOTO 1980
1970 Y1=R1
1980 GOSUB 2290
1990 RETURN
2000 REM  CHECKS FIRE STAT
2010 IF KBA=49 THEN 2020 ELS
E 2070
2020 CALL KEY(5,KEY2,STAT2)
2030 IF STAT2=1 THEN 2100
2040 IF STAT2=0 THEN 2100
2050 IF KEY2=83 THEN 2120
2060 IF KEY2=115 THEN 2120 E
LSE 2280
2070 CALL KEY(1,KEY2,STAT2)
2080 IF STAT2=0 THEN 2100
2090 GOTO 2130
2100 STATC=0
2110 GOTO 2280
2120 STATC=STATC+1
2130 IF STATC>CK THEN 2280
2140 IF R1=R2 THEN 2160
2150 GOTO 2280
2160 IF FD=1 THEN 2280
2170 FD=1
2180 STATC=0
2190 CALL SOUND(1000,-3,2)
2200 M1=C1-2
2210 FOR M=C2+2 TO C1-3
2220 M1=M1-1
2230 CALL HCHAR(R2,M1,140)
2240 CALL HCHAR(R2,M1,139)
2250 NEXT M
2260 GOSUB 3210
2270 L1=1
2280 RETURN
2290 REM  PRINTS BLUE TANK
2300 IF R1=Y1 THEN 2360
2310 CALL HCHAR(R1,C1,139,3)
2320 R1=Y1
2330 CALL HCHAR(R1,C1,128)
2340 CALL HCHAR(R1,C1+1,129)
2350 CALL HCHAR(R1,C1+2,130)
2360 RETURN
2370 REM  SCORES
2380 FD=0
2390 Q1=Q1+1
2400 IF LV<3 THEN 2450
2410 IF LV<5 THEN 2470
2420 IF LV<7 THEN 2490
2430 IF LV<9 THEN 2510
2440 IF LV<11 THEN 2530 ELSE
 2550
2450 S2=25
2460 GOTO 2560
2470 S2=50
2480 GOTO 2560
2490 S2=100
2500 GOTO 2560
2510 S2=150
2520 GOTO 2560
2530 S2=200
2540 GOTO 2560
2550 S2=250
2560 FOR Q=1 TO Q1
2570 S2=S2*2
2580 NEXT Q
2590 IF S2>1000 THEN 2600 EL
SE 2610
2600 S2=1000
2610 SC=SC+S2
2620 MSG$=STR$(SC)
2630 PL1=2
2640 PL2=20
2650 GOSUB 4170
2660 IF SC>=100000 THEN 2790
2670 IF SC>=90000 THEN 2820
2680 IF SC>=80000 THEN 2850
2690 IF SC>=70000 THEN 2880
2700 IF SC>=60000 THEN 2910
2710 IF SC>=50000 THEN 2940
2720 IF SC>=40000 THEN 2970
2730 IF SC>=30000 THEN 3000
2740 IF SC>=20000 THEN 3030
2750 IF SC>=10000 THEN 3060
2760 RESTORE 2770
```

```
2770 DATA 9,6,3,3,1,3,1
2780 GOTO 3080
2790 RESTORE 2800
2800 DATA 3,2,2,2,1,13,11
2810 GOTO 3080
2820 RESTORE 2830
2830 DATA 4,3,2,3,1,12,10
2840 GOTO 3080
2850 RESTORE 2860
2860 DATA 5,5,2,4,1,11,9
2870 GOTO 3080
2880 RESTORE 2890
2890 DATA 6,4,3,3,1,10,8
2900 GOTO 3080
2910 RESTORE 2920
2920 DATA 6,4,3,4,1,9,7
2930 GOTO 3080
2940 RESTORE 2950
2950 DATA 7,4,3,3,1,8,6
2960 GOTO 3080
2970 RESTORE 2980
2980 DATA 7,5,3,4,1,7,5
2990 GOTO 3080
3000 RESTORE 3010
3010 DATA 7,5,2,3,1,6,4
3020 GOTO 3080
3030 RESTORE 3040
3040 DATA 7,5,2,4,1,5,3
3050 GOTO 3080
3060 RESTORE 3070
3070 DATA 7,5,2,3,1,4,2
3080 READ K,J1,J2,SK,CK,TT,L
V
3090 GOSUB 1080
3100 RETURN
3110 REM  BLOW UP RED TANK
3120 CALL HCHAR(R1,C1,139,3)
3130 CALL SOUND(300,-6,5,280
,1)
3140 CALL HCHAR(R1,C1+1,131)
3150 CALL HCHAR(R1-1,C1,131,
3)
3160 CALL HCHAR(R1+1,C1,131,
3)
3170 CALL HCHAR(R1,C1,139,3)
3180 CALL HCHAR(R1-1,C1,139,
3)
3190 CALL HCHAR(R1+1,C1,139,
3)
3200 RETURN
3210 REM  BLOW UP BLUE TANK
3220 CALL HCHAR(R2,C2,139,3)
3230 CALL SOUND(300,-5,5,200
,1)
3240 CALL HCHAR(R2,C2+1,141)
3250 CALL HCHAR(R2-1,C2,141,
3)
3260 CALL HCHAR(R2+1,C2,141,
3)
3270 CALL HCHAR(R2,C2,139,3)
3280 CALL HCHAR(R2-1,C2,139,
3)
3290 CALL HCHAR(R2+1,C2,139,
3)
3300 RETURN
3310 REM  BUILDS DISPLAY
3320 CALL CLEAR
3330 FOR J=2 TO 24 STEP 22
3340 FOR I=5 TO 23 STEP 6
3350 CALL HCHAR(J,I,136)
3360 CALL HCHAR(J,I+1,137)
3370 CALL HCHAR(J,I+2,138)
3380 CALL HCHAR(J,I+3,128)
3390 CALL HCHAR(J,I+4,129)
3400 CALL HCHAR(J,I+5,130)
3410 NEXT I
3420 NEXT J
3430 I=2
3440 FOR J=4 TO 22 STEP 2
3450 CALL HCHAR(J,I,136)
3460 CALL HCHAR(J,I+1,137)
3470 CALL HCHAR(J,I+2,138)
3480 NEXT J
3490 I=29
3500 FOR J=4 TO 22 STEP 2
3510 CALL HCHAR(J,I,128)
3520 CALL HCHAR(J,I+1,129)
3530 CALL HCHAR(J,I+2,130)
3540 NEXT J
3550 REM   TANK ATTACK DATA
3560 RESTORE 3570
3570 DATA 5,9,132,1,5,10,132
,4,5,11,132,1,5,13,132,4
3580 DATA 5,14,132,1,7,14,13
2,1,5,15,132,4
3590 DATA 5,17,132,4,6,18,13
3,1,7,18,134,1,5,19,132,4
3600 DATA 5,21,132,4,6,22,13
2,2,5,23,132,1
```

```
3610 DATA 8,23,132,1,12,5,13
2,4,12,6,132,1,14,6,132,1
3620 DATA 12,7,132,4,12,9,13
2,1,12,10,132,4
3630 DATA 12,11,132,1,12,13,
132,1,12,14,132,4
3640 DATA 12,15,132,1,12,17,
132,4,12,18,132,1
3650 DATA 14,18,132,1,12,19,
132,4,12,21,132,4
3660 DATA 12,22,132,1,15,22,
132,1,12,23,132,1
3670 DATA 15,23,132,1,12,25,
132,4,13,26,132,2
3680 DATA 12,27,132,1,15,27,
132,1
3690 FOR I=1 TO 38
3700 READ A,B,C,D
3710 CALL VCHAR(A,B,C,D)
3720 NEXT I
3730 MSG$="HIT ANY KEY"
3740 PL1=18
3750 PL2=10
3760 GOSUB 4170
3770 MSG$="HI-"
3780 PL1=21
3790 PL2=5
3800 GOSUB 4170
3810 IF SC<SCM THEN 3830
3820 SCM=SC
3830 MSG$=STR$(SCM)
3840 PL1=21
3850 PL2=8
3860 GOSUB 4170
3870 MSG$="LAST-"
3880 PL1=21
3890 PL2=17
3900 GOSUB 4170
3910 MSG$=STR$(SC)
3920 PL1=21
3930 PL2=22
3940 SC=0
3950 GOSUB 4170
3960 CALL KEY(3,RT,SV)
3970 IF SV=0 THEN 3960
3980 RESTORE 3990
3990 DATA 9,9,4,1,3,0,3,1
4000 READ K,J1,J2,LV,TT,ST,S
K,CK
4010 RETURN
4020 REM  PART OF DISPLAY
4030 CALL CLEAR
4040 MSG$="HI SCORE "
4050 PL1=2
4060 PL2=1
4070 GOSUB 4170
4080 PL1=2
4090 PL2=11
4100 IF SC<SCM THEN 4120
4110 SCM=SC
4120 MSG$=STR$(SCM)
4130 GOSUB 4170
4140 RETURN
4150 REM  SCREEN PLACEMENT
4160 REM  OF MESSAGE
4170 M0=LEN(MSG$)
4180 FOR I=1 TO M0
4190 I$=SEG$(MSG$,I,1)
4200 M9=ASC(I$)
4210 CALL HCHAR(PL1,I+PL2,M9
)
4220 NEXT I
4230 RETURN
```

HAPPY COMPUTING!

```
************************
*    BUILDING BLOCKS   *
*       V-PB131KB      *
*     BY T CASTLE      *
************************
```

DESCRIPTION. Building Blocks is a simple yet entertaining program for youngsters from 4 to 10 years of age. Upon entering the RUN command, the child is greeted with a delightful display of shapes and colors. The screen color is Cyan and, across the top of the screen, in a band three rows high, the child finds a white grid with gray lines. Within this band, superimposed on the grid, there are three differently shaped objects, in three different sizes each, presented in various colors. There are three triangles, three circles, and three squares, each in large, medium, and small. Each of the objects is lettered A through I. To the left side of the screen there are four colored blocks: green, yellow, red, and blue. These blocks are labeled A through D. Below these there is another white grid pattern labeled "E" and the word "NEW" labeled "F". Consuming the major portion of the screen there is a large white grid with gray lines. Down the left side of the grid it is labeled "ROW" and each line is lettered A through P. Across the top of the grid the word "COLUMN" appears and it too is lettered A through S. Below the blocks on the left side of the screen the words "ENTER COLOR" appear and a "beeping", "blinking" question mark flashes on and off. When the child selects a color and enters a letter such as "A" for green, the word "GREEN" is displayed below the grid, and the question is changed to "ENTER SHAPE". When the letter representing the shape and size is entered, the message across the bottom is completed; e.g. "LARGE GREEN TRIANGLE". The child then enters a letter for "ROW" and a letter for "COLUMN". The row and column should represent the lower left hand corner of the shape selected. Upon completion, the child is given a pleasant "ting-a-ling" and the object is placed on the grid. By continuing in this manner, the child can build all sorts of interesting objects such as rocket ships, cars, or just plain designs. If a mistake is made he can use the selection "E" (grid pattern) to clear just one item or a portion of an item from the screen. If the "F" is pushed for "NEW" then the entire grid is erased and ready for a new pattern.

NOTES. The general sequence of this program is layed out in lines 160 through 300. A refined and condensed version of this program also appears at the end of Chapter 10. While this version is longer in terms of line numbers, it is far easier to understand and debug if you should make a mistake entering it. It is also more suitable for modifications and revisions if you want to try some other ideas.

The following sequence builds a simple rocket ship: B,A,C,I; D,G,F,I; D,G,I,I; B,A,N,G; B,B,L,H; B,A,N,K; B,B,L,K; D,G,L,I; E,G,O,I.

```
100 REM  *****************
110 REM  *BUILDING BLOCKS*
120 REM  *****************
130 REM  BY T CASTLE
140 REM  AMLIST V-PB131KB
150 REM
160 CALL CLEAR
170 GOSUB 310
180 GOSUB 1240
190 GOSUB 610
200 GOSUB 2320
210 GOSUB 2550
220 IF A1=70 THEN 190
230 GOSUB 2270
240 GOSUB 2870
250 GOSUB 2370
260 GOSUB 3250
270 GOSUB 2420
280 GOSUB 3490
290 GOSUB 3730
300 GOTO 200
310 CALL SCREEN(8)
320 REM  DEFINE COLOR SETS
330 RESTORE 340
340 DATA 9,13,10,13,11,11,12
,11
350 DATA 13,9,14,9,15,5,16,5
,2,15
360 FOR I=1 TO 9
370 READ A,B
380 CALL COLOR(A,B,16)
390 NEXT I
400 REM  DEFINES CHAR SHAPES
410 DATA 030F3F3F7F7FFFFF,FF
FF7F7F3F3F0F03
420 DATA C0F0FCFCFEFEFFFF,FF
FFFEFEFCFCF0C0
430 DATA 0001071F1F3F3F7F,7F
3F3F1F1F070100
440 DATA 0080E0F8F8FCFCFE,FE
FCFCF8F8E08000
450 DATA FFFFFFFFFFFFFFFF,18
183C3C7E7EFFFF
460 DATA 0101030307070F0F,1F
1F3F3F7F7FFFFF
470 DATA 8080C0C0E0E0F0F0,F8
F8FCFCFEFEFFFF
480 DATA 3C7EFFFFFFFF7E3C
490 START=80
500 FOR I=1 TO 4
510 RESTORE 410
520 START=START+16
530 FOR K=START TO START+14
540 READ A$
550 CALL CHAR(K,A$)
560 NEXT K
570 NEXT I
580 CALL CHAR(40,"FF81818181
8181FF")
590 RETURN
600 REM   BLDS BACKGROUND
610 FOR J=8 TO 23
620 CALL HCHAR(J,13,40,19)
630 NEXT J
640 RETURN
650 REM  BLD 1X1 SCREEN BLK
660 CALL HCHAR(R1,C1,40)
670 RETURN
680 REM  BLD 2X2 SCREEN BLK
690 CALL HCHAR(R1-1,C1,40,2)
700 CALL HCHAR(R1,C1,40,2)
710 RETURN
720 REM  BLD 3X3 SCREEN BLK
730 CALL HCHAR(R1-2,C1,40,3)
740 CALL HCHAR(R1-1,C1,40,3)
750 CALL HCHAR(R1,C1,40,3)
760 RETURN
770 REM  LG TRI
780 CALL HCHAR(R1,C1,CC)
790 CALL HCHAR(R1+1,C1-1,CC+
1)
800 CALL HCHAR(R1+2,C1-1,CC+
2)
810 CALL HCHAR(R1+1,C1+1,CC+
3)
820 CALL HCHAR(R1+2,C1+1,CC+
4)
830 CALL VCHAR(R1+1,C1,CC-1,
2)
840 RETURN
850 REM  MED TRI
860 CALL HCHAR(R1,C1,CC+1)
870 CALL HCHAR(R1+1,C1,CC+2)
880 CALL HCHAR(R1,C1+1,CC+3)
890 CALL HCHAR(R1+1,C1+1,CC+
4)
900 RETURN
910 REM  MED BL
920 CALL HCHAR(R1,C1,CC-9)
930 CALL HCHAR(R1+1,C1,CC-8)
```

```
940 CALL HCHAR(R1,C1+1,CC-7)
950 CALL HCHAR(R1+1,C1+1,CC-
6)
960 RETURN
970 REM  LG BL
980 CALL HCHAR(R1,C1,CC-5)
990 CALL HCHAR(R1+2,C1,CC-4)
1000 CALL HCHAR(R1,C1+2,CC-3
)
1010 CALL HCHAR(R1+2,C1+2,CC
-2)
1020 CALL HCHAR(R1+1,C1,CC-1
,3)
1030 CALL VCHAR(R1,C1+1,CC-1
,3)
1040 RETURN
1050 REM  MED BLOCK
1060 CALL HCHAR(R1,C1,CC-1,2
)
1070 CALL HCHAR(R1+1,C1,CC-1
,2)
1080 RETURN
1090 REM  LG BLOCK
1100 CALL HCHAR(R1,C1,CC-1,3
)
1110 CALL HCHAR(R1+1,C1,CC-1
,3)
1120 CALL HCHAR(R1+2,C1,CC-1
,3)
1130 RETURN
1140 REM  BLDS SINGLE BLOCK
1150 CALL HCHAR(R1,C1,CC-1)
1160 RETURN
1170 REM  BLDS SINGLE BALL
1180 CALL HCHAR(R1,C1,CC+5)
1190 RETURN
1200 REM  BLDS SINGLE TRI
1210 CALL HCHAR(R1,C1,CC)
1220 RETURN
1230 REM  STARTING DISPLAY
1240 FOR J=1 TO 3
1250 CALL HCHAR(J,3,40,29)
1260 NEXT J
1270 R1=1
1280 C1=6
1290 CC=121
1300 GOSUB 780
1310 R1=2
1320 C1=9
1330 CC=105
1340 GOSUB 860
1350 R1=3
1360 C1=12
1370 CC=153
1380 GOSUB 1210
1390 R1=1
1400 C1=14
1410 CC=137
1420 GOSUB 980
1430 R1=2
1440 C1=18
1450 CC=121
1460 GOSUB 920
1470 R1=1
1480 C1=23
1490 CC=105
1500 GOSUB 1100
1510 R1=3
1520 C1=21
1530 CC=153
1540 GOSUB 1180
1550 R1=2
1560 C1=27
1570 CC=137
1580 GOSUB 1060
1590 R1=3
1600 C1=30
1610 CC=121
1620 GOSUB 1150
1630 R1=7
1640 C1=4
1650 CC=105
1660 GOSUB 1060
1670 R1=7
1680 C1=7
1690 CC=121
1700 GOSUB 1060
1710 R1=11
1720 C1=4
1730 CC=137
1740 GOSUB 1060
1750 R1=11
1760 C1=7
1770 CC=153
1780 GOSUB 1060
1790 R1=16
1800 C1=4
1810 GOSUB 690
1820 RESTORE 1830
1830 DATA 4,5,65,4,9,66
```

```
1840 DATA 4,12,67,4,14,68
1850 DATA 4,18,69,4,21,70
1860 DATA 4,23,71,4,27,72
1870 DATA 4,30,73
1880 DATA 9,4,65,9,7,66
1890 DATA 13,4,67,13,7,68
1900 DATA 17,4,69,15,7,78
1910 DATA 15,8,69,15,9,87
1920 DATA 17,7,70
1930 DATA 67,79,76,85,77,78,
82,79,87
1940 FOR I=1 TO 18
1950 READ A,B,C
1960 CALL HCHAR(A,B,C)
1970 NEXT I
1980 FOR I=13 TO 31
1990 CALL HCHAR(7,I,I+52)
2000 NEXT I
2010 FOR I=19 TO 24
2020 READ A
2030 CALL HCHAR(6,I,A)
2040 NEXT I
2050 FOR I=14 TO 16
2060 READ A
2070 CALL HCHAR(I,11,A)
2080 NEXT I
2090 FOR I=8 TO 23
2100 CALL HCHAR(I,12,I+57)
2110 NEXT I
2120 GOSUB 2220
2130 RETURN
2140 REM   PRINTS MSG
2150 MSGL=LEN(MSG$)
2160 FOR I=1 TO MSGL
2170 I$=SEG$(MSG$,I,1)
2180 MSGC=ASC(I$)
2190 CALL HCHAR(R1,I+C1,MSGC
)
2200 NEXT I
2210 RETURN
2220 MSG$="ENTER "
2230 R1=20
2240 C1=3
2250 GOSUB 2150
2260 RETURN
2270 MSG$="SHAPE "
2280 R1=21
2290 C1=3
2300 GOSUB 2150
2310 RETURN
2320 MSG$="COLOR "
2330 R1=21
2340 C1=3
2350 GOSUB 2150
2360 RETURN
2370 MSG$="ROW   "
2380 R1=21
2390 C1=3
2400 GOSUB 2150
2410 RETURN
2420 MSG$="COLUMN"
2430 R1=21
2440 C1=3
2450 GOSUB 2150
2460 RETURN
2470 R1=22
2480 C1=3
2490 MSG$="?     "
2500 CALL SOUND(5,1175,1)
2510 GOSUB 2150
2520 CALL HCHAR(R1,C1,32,2)
2530 RETURN
2540 REM  CALL KEY FOR COLOR
2550 GOSUB 2470
2560 CALL KEY(3,A1,STAT)
2570 IF STAT=0 THEN 2550
2580 IF A1>64 THEN 2590 ELSE
 2550
2590 IF A1<71 THEN 2600 ELSE
 2550
2600 CALL HCHAR(24,9,32,24)
2610 R1=22
2620 C1=7
2630 MSG$=CHR$(A1)
2640 GOSUB 2150
2650 IF A1=65 THEN 2660 ELSE
 2680
2660 CLR$=" GREEN "
2670 GOTO 2810
2680 IF A1=66 THEN 2690 ELSE
 2710
2690 CLR$=" YELLOW "
2700 GOTO 2810
2710 IF A1=67 THEN 2720 ELSE
 2740
2720 CLR$=" RED "
2730 GOTO 2810
2740 IF A1=68 THEN 2750 ELSE
 2770
2750 CLR$=" BLUE "
```

```
2760 GOTO 2810
2770 IF A1=69 THEN 2780 ELSE
 2800
2780 CLR$=" CLEAR "
2790 GOTO 2810
2800 CLR$="        "
2810 MSG$=CLR$
2820 R1=24
2830 C1=11
2840 GOSUB 2140
2850 RETURN
2860 REM  CALL KEY FOR SHAPE
2870 GOSUB 2470
2880 CALL KEY(3,A2,STAT)
2890 IF STAT=0 THEN 2870
2900 IF A2>64 THEN 2910 ELSE
 2970
2910 IF A2<74 THEN 2920 ELSE
 2870
2920 R1=22
2930 C1=7
2940 MSG$=CHR$(A2)
2950 GOSUB 2150
2960 IF A2=65 THEN 2970 ELSE
 2990
2970 MSG$="LARGE"&CLR$&"TRIA
NGLE"
2980 GOTO 3210
2990 IF A2=66 THEN 3000 ELSE
 3020
3000 MSG$="MEDIUM"&CLR$&"TRI
ANGLE"
3010 GOTO 3210
3020 IF A2=67 THEN 3030 ELSE
 3050
3030 MSG$="SMALL"&CLR$&"TRIA
NGLE"
3040 GOTO 3210
3050 IF A2=68 THEN 3060 ELSE
 3080
3060 MSG$="LARGE"&CLR$&"CIRC
LE"
3070 GOTO 3210
3080 IF A2=69 THEN 3090 ELSE
 3110
3090 MSG$="MEDIUM"&CLR$&"CIR
CLE"
3100 GOTO 3210
3110 IF A2=70 THEN 3120 ELSE
 3140
3120 MSG$="SMALL"&CLR$&"CIRC
LE"
3130 GOTO 3210
3140 IF A2=71 THEN 3150 ELSE
 3170
3150 MSG$="LARGE"&CLR$&"SQUA
RE"
3160 GOTO 3210
3170 IF A2=72 THEN 3180 ELSE
 3200
3180 MSG$="MEDIUM"&CLR$&"SQU
ARE"
3190 GOTO 3210
3200 MSG$="SMALL"&CLR$&"SQUA
RE"
3210 R1=24
3220 C1=9
3230 GOSUB 2150
3240 RETURN
3250 REM  CALL KEY FOR ROW
3260 LM=0
3270 GOSUB 2470
3280 CALL KEY(3,A3,STAT)
3290 IF STAT=0 THEN 3270
3300 IF A2=65 THEN 3330
3310 IF A2=68 THEN 3330
3320 IF A2=71 THEN 3330 ELSE
 3350
3330 LM=2
3340 GOTO 3410
3350 IF A2=66 THEN 3380
3360 IF A2=69 THEN 3380
3370 IF A2=72 THEN 3380 ELSE
 3400
3380 LM=1
3390 GOTO 3410
3400 LM=0
3410 IF A3>64+LM THEN 3420 E
LSE 3270
3420 IF A3<81 THEN 3430 ELSE
 3270
3430 R1=22
3440 C1=7
3450 MSG$=CHR$(A3)
3460 GOSUB 2150
3470 RETURN
3480 REM  CALL KEY FOR COLMN
3490 LM=0
3500 GOSUB 2470
3510 CALL KEY(3,A4,STAT)
```

```
3520 IF STAT=0 THEN 3500
3530 IF A2=65 THEN 3560
3540 IF A2=68 THEN 3560
3550 IF A2=71 THEN 3560 ELSE
 3580
3560 LM=2
3570 GOTO 3640
3580 IF A2=66 THEN 3610
3590 IF A2=69 THEN 3610
3600 IF A2=72 THEN 3610 ELSE
 3630
3610 LM=1
3620 GOTO 3640
3630 LM=0
3640 IF A4>64 THEN 3650 ELSE
 3500
3650 IF A4<84-LM THEN 3660 E
LSE 3500
3660 R1=22
3670 C1=7
3680 MSG$=CHR$(A4)
3690 GOSUB 2150
3700 RETURN
3710 REM  THIS CALCULATES
3720 REM  AND PRINTS SHAPE
3730 CALL SOUND(15,1319,1)
3740 CALL SOUND(15,1109,1)
3750 CALL SOUND(15,1319,1)
3760 CALL SOUND(15,1109,1)
3770 CALL SOUND(25,1319,1)
3780 IF A1=65 THEN 3830
3790 IF A1=66 THEN 3850
3800 IF A1=67 THEN 3870
3810 IF A1=68 THEN 3890
3820 IF A1=69 THEN 3910
3830 CC=105
3840 GOTO 4060
3850 CC=121
3860 GOTO 4060
3870 CC=137
3880 GOTO 4060
3890 CC=153
3900 GOTO 4060
3910 CC=0
3920 R1=A3-57
3930 C1=A4-52
3940 IF A2=65 THEN 3970
3950 IF A2=68 THEN 3970
3960 IF A2=71 THEN 3970 ELSE
 3990
3970 GOSUB 730
3980 GOTO 4050
3990 IF A2=66 THEN 4020
4000 IF A2=69 THEN 4020
4010 IF A2=72 THEN 4020 ELSE
 4040
4020 GOSUB 690
4030 GOTO 4050
4040 GOSUB 660
4050 GOTO 4530
4060 R2=A3-57
4070 C2=A4-52
4080 IF A2=65 THEN 4170
4090 IF A2=66 THEN 4210
4100 IF A2=67 THEN 4250
4110 IF A2=68 THEN 4290
4120 IF A2=69 THEN 4330
4130 IF A2=70 THEN 4370
4140 IF A2=71 THEN 4410
4150 IF A2=72 THEN 4450
4160 IF A2=73 THEN 4490
4170 R1=R2-2
4180 C1=C2+1
4190 GOSUB 780
4200 GOTO 4530
4210 R1=R2-1
4220 C1=C2
4230 GOSUB 860
4240 GOTO 4530
4250 R1=R2
4260 C1=C2
4270 GOSUB 1210
4280 GOTO 4530
4290 R1=R2-2
4300 C1=C2
4310 GOSUB 980
4320 GOTO 4530
4330 R1=R2-1
4340 C1=C2
4350 GOSUB 920
4360 GOTO 4530
4370 R1=R2
```

```
4380 C1=C2
4390 GOSUB 1180
4400 GOTO 4530
4410 R1=R2-2
4420 C1=C2
4430 GOSUB 1100
4440 GOTO 4530
4450 R1=R2-1
4460 C1=C2
4470 GOSUB 1060
4480 GOTO 4530
4490 R1=R2
4500 C1=C2
4510 GOSUB 1150
4520 GOTO 4530
4530 A1=0
4540 A2=0
4550 A3=0
4560 A4=0
4570 RETURN
```

HAPPY COMPUTING!

CHAPTER THREE

Debugging & Error Messages

GENERAL. By now most of you have had an opportunity to enter or key in at least one or two of the programs found in this manual or at least from one of the magazines. Unless you're "one in a million" you've also come up against what is affectionately know as a "bug" or an error in the program. This chapter is devoted to a discussion of the "bug", in it's several varieties, and to the error messages in general. Before we go any further, let's begin by making a couple of general rules:

RULE 1 - YOU HAVE MADE A MISTAKE!

If you start with this assumption, you'll generally save yourself many hours of grief and worry over the system or the program you're working on. That's not to say that the computer won't ever go bad on you; however, if it does, it'll be unmistakable and won't affect a single action, such as one particular subroutine or one FOR-NEXT loop. Further, while it's possible that printed programs, such as ours or those found in other publications, have a flaw in them, it's not as likely as the original assumption. Our programs and other printed programs are tested many times in a number of ways prior to their being released. Sometimes newly released programs will run without error for days, weeks, months, and even years, and then without warning they will "error out". For instance, this could occur in a game program which has a series of IF-THEN statements following the scoring routine. Perhaps there are branching statements for 1Ø,ØØØ, 2Ø,ØØØ, 3Ø,ØØØ points and there's an error in the 3Ø,ØØØ statement. If nobody ever reaches that level, the program will always branch out prior to hitting that line and no error will ever be detected. Until you can identify an error in a printed program specifically, meaning that you know what line it's in, what's wrong with it, and how to correct it -- stick with the rule that "You Have Made a Mistake". Having swallowed that bitter pill, let's go on to the second rule.

RULE 2 - TEST PROGRAMS FREQUENTLY!

Did you ever try to find a $12.ØØ error in a checkbook that hasn't been balanced in a year. It can be a very time consuming process since there are so many entries to verify and the error may actually be a combination of errors all adding up to $12.ØØ. This is an easier rule to follow when writing your own programs than it is when copying others, but with a little practice it can be applied in both cases. The trick is to isolate a particular portion of the program, using temporary program lines such as GOTO's and DATA statements, and to run it like a miniature program. We'll give specific examples on how to do this later, but the point is it's like balancing each months bank statement. If you've checked each subroutine individually, and you've only entered 2Ø lines since it was last checked, any technical error must be in the last section entered.

Definitions. There are really three major kinds of errors or problems with programs which are called "bugs". The first of these is what we call a "Technical Bug". The misspelling of a command word such as PRINT or GOSUB is an example. The result of a technical bug is an error message immediately after hitting the ENTER key or when the program begins to run. For lack of a better name we're going to call the second type a "Programming Bug". This is the type referred to earlier, where a program runs for some time without error and then, when a certain condition exists, it produces an error message. The third type is a "Logic Bug" and never really produces and error message. An example would be a game program that's supposed to end when the last of three buildings is bombed. If the third building was bombed, but the game kept on going, allowing you to continue to score points, this would be a logic bug. No error message would ever be produced, yet the program isn't doing what it's supposed to do. Each of these has unique characteristics, and the methods involved for preventing them, finding them, and correcting them deserve separate treatment.

Technical Bug. The User's Reference Guide is your best aid in finding a technical bug. The computer is checking your program constantly for errors. In fact, it really checks it at three different times. The first check is made immediately after you've keyed in a line and hit the "ENTER" key. Since the error message displayed can only be in the last line entered, these are easy to define. Usually the error is obvious to the user; however, if it's still not clear, refer to the section "Errors Found When Entering a Line" in the URG and compare your entry with the list of things that can cause the error.

After you've keyed in a program and you type the command word RUN, before the program actually begins going through the lined statements, it performs a "pre-check" or scan of the program for other types of errors. This is where debugging can become a little tricky because the error messages are sometimes misleading and/or they don't give you a clue as to where the error is. When you encounter one of these errors, refer to the URG as your first choice. The URG states that these errors also indicate the line where the error is found. As best we can determine, this is always so, with the exception of the FOR-NEXT ERROR. During the scan the computer keeps track of how many FOR statements there are and how many NEXT statements there are. When the scan is completed, if they are not equal, an error message will be displayed, but it can't tell you which FOR statement is missing the NEXT statement. To demonstrate what can happen with FOR-NEXT statements, enter the following small program. It should print "TEST 1" ten times and then say ** DONE **:

```
>10 FOR I=1 TO 10
>20 PRINT "TEST 1"
>30 NEXT I
>40 STOP
```

Now we'll show you four errors that could come up in this type of statement. First, remove line 30 (type 30 and hit the ENTER key). When you RUN the program you will get the error message "* FOR-NEXT ERROR", with no line reference. Second, replace line 30 with the ">30 NEXT J". RUNning this will produce the error

message "*CAN'T DO THAT IN 30". Third, enter ">30 NEXT". This produces the error message "*INCORRECT STATEMENT IN 30". Lastly, replace line 30 with the correct statement, i.e. "30 NEXT I" and add ">35 GOTO 20". This again will produce the "*CAN'T DO THAT IN 30" message because you can't do a "branch" into the middle of a FOR-NEXT loop.

All of these were really FOR-NEXT errors, yet only one produced that message. If you look up each of the other error messages in your user's guide, you'll find the solution to each problem. It doesn't specify FOR-NEXT in each case, because other things can also cause these problems. Except for the first error, you should be able to "debug" these with the line number reference. If you get the first error message, in a full program with 5 or 10 FOR-NEXT loops, you're only clue will be what's happening in the program at that time, i.e.:the "opening display" is being created; the "gun" is being fired; the program should be sorting; etc. Review your line listing for the portion of the program that controls that activity, looking specifically for FOR-NEXT loops. The way to prevent this in the first place is to test your program frequently as you enter it.

If, after reviewing the error messages, you still can't see what's wrong, you'll have to begin considering other possibilities. In the "KAMAZAZE RUN" program, if you remove the word CALL from line 2260 CALL GCHAR(23,8,BL1), and RUN the program, you'll get a MEMORY FULL error message in line 2260. In fact, this message will appear anytime you remove the word CALL before any GCHAR, HCHAR, or VCHAR in this program. Of course, it's really an INCORRECT STATEMENT that has caused a MEMORY FULL condition. Later in this chapter we'll show you how to check available memory and, as you become more adept at programming, you'll automatically have a feel for whether you're really out of memory or if there's some other problem causing the condition.

Incidentally, just because the missing CALL produced a MEMORY FULL condition in the above example, it may not always do so. Type the following line:

```
>10 HCHAR(10,10,68)
```

Now RUN this program. You'll get the error message BAD SUBSCRIPT IN 10. Again, there's nothing wrong with the subscript, it's an INCORRECT STATEMENT. This brings us to another type of technical bug, the kind that's found when the program actually begins to sequence through the numbered statements.

It's hard to put a percentage on it, but let's just say "frequently", the error message given and the line referenced don't indicate the actual problem. While doing research for this manual, and while "debugging" programs over the phone, by a 9 to 1 margin, the most frequent error reported was "BAD VALUE IN (some line number)". Look at page 1 of the "TANK ATTACK" line listing and let's discuss the possible problems. If the line itself is correct, then the bad value created must be the A or A$ value. The first thing to do when you encounter this error is to use the "command mode" to print the value of each variable involved. In this case you would tell the computer to PRINT A and PRINT A$. Compare these with the data statements from which it's reading. Let's say that A is 139 and

A$ is filled with all "O's" instead of "zeros". At that point, your error is obvious and can be corrected. If you get a zero value for A, the error is probably in the READ statement in line 750. If you typed in READ B,A$, you would get a "0" for A and "3F010FFFFF3F1F0F" for A$. As a last example, if line 740 read "FOR I=1 to 14" instead of 13, the value of A would be 3 and A$ would be 2. By reviewing the program you can see that, after it took the first 13 lines of data, it would begin working on line 730 and get the first two items as A and A$.

It's impossible to point out all of the combinations of errors that can be generated in a running program. Unlike errors reported in the scanning process, in a running program we get involved with a number of variables that are being created and interact with each other. These sometimes produce misleading error statements and line references. Hopefully, the above example will help you with the thought process involved in tracing these errors. The point is, if you can't define the problem based on the explanation in the URG, consider all other possibilities, such as; incorrect statements, misspelled words, or bad values for the variables involved. Now let's move on to another type of bug.

Programming Bug. This is one of the worst types of bugs to have (yet we all get them) because the first thing you have to do is realize that there is a bug and then locate the problem. In order to discuss this, we're going to have to talk about programs you may copy, as opposed to your own written programs, because the approach differs slightly. Let's take copied programs first.

First, there's no law that says you have to enter a program in consecutive line number order, beginning with the first line and continuing through to the last line; and second, debugging is easier if it's done gradually, throughout the process of entering the program, rather than as a single step when completed. When you get a program that you want to copy, the first thing to do is read what it's supposed to do; and the second is to look over the line listing and familiarize yourself with the layout. Our programs are generally broken down into well defined subroutines, while others may not be. In either case, try to get some understanding of the organization before you begin.

We're going to use the "KAMAKAZE RUN" program, found at the end of this chapter, as an example of how to enter a program and debug it as you go along. Begin entering this program by keying in lines 100-160, the first branching statement where the program is sent to a subroutine (GOSUB 320). Now type in:

```
>165 PRINT "OK TO HERE"
>166 STOP
```

Now enter lines 310 through 620 (the RETURN statement). (NOTE: You'll always find that our GOSUB's reference the first "active" line of a subroutine and not the REMark that goes with it. This enables us to add or remove REMarks at anytime without having to go back and change the branching statements.) After these lines are entered, type the RUN command. If you have any technical errors such as O's for zeros or a missing parenthesis, you'll know it now and you won't be surprised later. Next, remove 166 and 167, and then enter line 170 (GOSUB 640). Add

another temporary stop using lines 175-176. Now enter 63Ø through 97Ø. Again, RUN the program. You'll now get an error message that says "BAD LINE NUMBER IN 85Ø". We've referenced a subroutine that hasn't been keyed in yet. Go to that point in the program and key in 293Ø to 3ØØØ and then RUN the program again. The same thing will happen in lines 86Ø and 93Ø. Each time you encounter one, enter the subroutine and RUN the program. Keep doing this until you come back to your "OK TO HERE" message. At this point it might be worth your effort to SAVE the program that you've just created. It should be fully functional and error free to this point.

We're not going to go through this entire program, but the pattern should be evident. We suggest you enter the program in the order in which it runs, not necessarily line number order. By doing this you're keeping an operational program going at all times and any errors or "bugs" that develop must be in the last section entered. This makes finding them and correcting them a great deal easier. Of course, you could enter the entire program and then work through it on the same basis, but this means that you may have to spend 3, 4, or 5 hours before you begin to see progress. By doing it in the manner we suggest, the program starts to take shape almost immediately and provides you with some immediate reward for your effort.

Now let's discuss how this differs from a program which you write yourself. We're going to use the "KAMAKAZE RUN" program as an example again and give you the general sequence of events which took place in the writing of this program. This may seem to be a digression from the "debugging" topic, but it's our philosophy that half of the debugging battle is knowing where an error occurred. By keeping each step small and RUNning the program frequently we keep the possibilities to a minimum.

We started, as suggested in Chapter 2, with the main movement of the program, the dropping planes. We set up one array, A(12), and assigned a value of 3 to each element of the array. Next, on row 3 of the screen, we used the CALL HCHAR command to print a solid block in every other space from 2 to 24. We then used the RANDOMIZE and RND commands to select a number from 1 to 12, representing which plane we wanted to move, and increased the value of that element of the array by 1. For instance, if the computer selected 6, then A(6) would equal 4. Based on the element of the array it was possible to calculate the COLUMN in which we would move and the value of A equaled the ROW where we wanted to display the new plane. Again, we used a CALL HCHAR to print a new plane, 1 row down from the original, and then we erased the old plane. We then cycled it back through the RND command and let the program run. What happened was that 12 blocks randomly dropped down the screen until the value recorded in A exceeded 24, at which point the computer errored out. Since we were satisfied with the speed of the movement and the overall effect, we proceeded to write a program based on this technique.

Following is the way that this program was broken down and the sequence in which it was written. After each section, the program was RUN and tested to our complete satisfaction before the next step was taken.

1. A subroutine was set up at line 1ØØØ for initial variables. The only things originally included were the statements to: DIMension two arrays with twelve elements; CLEAR the screen; turn the screen black; and define the shape of a plane in two sets (one yellow and one red). A second subroutine was set up at line 2ØØØ to print 24 planes to the screen in two staggered rows across the top. Lines 1Ø, 2Ø, and 3Ø sent the program through GOSUB 1ØØØ, GOSUB 2ØØØ, and then stopped in 3Ø. The program was tested completely at this point.

2. Next we defined the necessary characters and defined the character sets to the appropriate colors to create the green band, a blue gunship, and the white buildings. We added these in the subroutine beginning at 1ØØØ. We then added to the subroutine at 2ØØØ to give us a complete display. At this point, there was no movement and no numbers appeared for scoring, "Hit Any Key", or "Pause".

3. A subroutine was set up at 4ØØØ to move the red row randomly down the screen. There were no checks for end of game or bottom of screen. At this point it just errored out.

4. A subroutine was built at 5ØØØ to move the red row off of the screen to the left before it got to the level of the blue ship. A line was added to the 4ØØØ routine to check its position after each movement and send it to 5ØØØ if it had reached a point low enough on the screen.

5. When the above two routines were working, we modified them slightly and entered them at 3ØØØ and 6ØØØ to handle the yellow planes. An additional check point was added to this routine to make sure the yellow planes didn't move below the level of the red planes.

6. A movement routine using the CALL KEY was created at line 7ØØØ to permit: movement and firing of the blue gunship; erasing of the ship if it was in line with the firing; and replacement of its spot with a blank space.

7. A routine was created at 8ØØØ to bomb the buildings when the planes passed over and characters were redefined to create rubble on the ground.

8. Check routines were added to appropriate subroutines to check if all 24 planes had been destroyed or if all 3 buildings were bombed. Appropriate statements were added to build a new display or end the program.

9. Scoring routines were added after each point where a plane was hit and a subroutine was built to print the score to the green band.

1Ø. CALL KEY statements were added and subroutines built for "PAUSE" and "HIT ANY KEY". An additional subroutine was built to print the "HI SCORE".

11. SOUND affects were added at appropriate points.

The above example shows how a program is built, piece by piece, while checking, running, and debugging throughout its creation. By building

it in this fashion, few problems are insurmountable and programming errors are easy to find and correct.

Logic Bug. If you have copied or created a program in the order and fashion outlined above, you may have already "caught" a lot of the logic bugs. Still, you may have gotten all the way through it and have a program that appears to function properly in every way, and you may still have problems. The final thing to do is to test it in every conceivable way.

In developing the "BASEBALL STATS" program in Chapter 7, we created a string, six characters long, that represented each boy's game statistics. When this program was finished we wrote a separate subroutine, at the beginning, to randomly create statistics for all twelve boys for all sixteen games. We bypassed the routine that filled these slots with zeros. We then ran the accumulator routine to calculate all the statistics, SAVEd it, and then loaded it again. Then we changed some of the stats, SAVEd it again, and loaded it again. So far, we had no errors. We then ran the program with the zero balances for each player and ran it through the option which calculates averages, percentage on base, etc. At this point the program errored out in line 2210. We were trying to divide by zero. A special line was added at 2200 to bypass 2210 if the value was zero. RUNning it again resulted in an error in 2340. This was the same problem.

In a game program, the method of testing may be different. In the "KAMAKAZE RUN" program the scoring routine is based on the value of LVL. LVL also is the number of rows that each plane moves on each movement. The LVL value is initially set in line 400. By increasing this to 3, 4, or 5, you can cause the program to skip boards and test its reaction at higher scoring levels. Additional errors are sometimes found using this method. In the "TANK ATTACK" program scoring and other changes take place in the program based on the score. Find the initial starting value and put in a temporary line with a high score and see what happens.

Available Memory. There's a subroutine at the end of the "BASEBALL STATS" program running from 3980 to 4020. It is very handy and quite accurate. In place of the 7.9787... you could just use a value of 8 and it will be close enough for most applications. We knew the "BASEBALL STATS" program would eat up a lot of memory with the creation of the arrays so we added this routine at line 9000 very early in the programming. After we created the arrays, we put in a temporary line that said GOTO 9000. After running the program and waiting a few seconds, the program would error out. At that point we typed in:

```
>PRINT FREMEM
```

The computer would return a value representing the approximate number of bytes available in memory.

Resequencing. Since this manual was designed for those who have only the straight 99/4A, without a printer or other peripherals, we recommend that you write your programs in large blocks (1000, 2000, 3000, etc). This leaves plenty of room for expansion and, more importantly, you can easily

remember where significant portions of your program reside. If you have a printer you can resequence more frequently and print out your latest version. If you need to know where a subroutine starts you can find it quicker on a piece of paper than on the screen. If you're copying from a program you'll usually already be working from a sequenced list. When you think you're finished with it, SAVE the program that you have in memory and then RESEQUENCE that program. Check the last line of the resequenced program against the last line of the printed program and they should agree. If they don't, randomly check line numbers until you find the point at which they don't agree and you'll find the line that you missed.

ENTER Key. Since the screen display is not extremely wide, you'll frequently have statements that will reach the end of the first line. If they fall directly on it, and you glance away from the screen for a second, it's easy to forget to hit the ENTER key. If you get an error message in a line, list a few lines above to a few lines below the referenced line. If you did forget the ENTER, one line will usually be offset to the right by one space.

RETYPE the Line. If you've tried everything else to find an error in a line, simply retype it. The "BUILDING BLOCKS" program has a series of lines in it that are IF-THEN-ELSE statements (see line 265Ø). The last letter of ELSE falls directly at the end of the line and the line number needs to go on the next line. If you glance at the written program to see what line to send it to, and then look back at the screen, it's easy to forget to type in the "space" between ELSE and the number. By simply retyping the line carefully you'll see a difference which isn't obvious by just listing the program.

BREAK and TRACE. Read up on the purpose of these two commands in the URG. The BREAK command is most often used before and after a particular part of the program that you can't get to function properly. Perhaps it's a scoring routine that utilizes two variables to arrive at a value for the score. You can put a break in just before the calculation and just after the line. At each break, use the command mode to print the value of the variables. By comparing these figures you may be able to figure out what's wrong with the calculation. If it's a multiple line calculation or a series of IF statements, the TRACE command can be helpful. If you don't have a printer, be sure to put a break point in shortly after the TRACE begins. It generates numbers very quickly and you may have to copy them down and then list that portion of the program to see what happened. Both the BREAK and TRACE command are of little value when trying to debug a game program with a lot of screen displays and redefined characters since it destroys the integrity of the display. Use the next option to take care of that problem.

Special Print Routines. In the "KAMAKAZE RUN" program we found that as the number of red or yellow planes were shot down, the remaining ones moved less and less frequently. The reason was that the computer was taking a RANDOM number from 1 to 12 and, if that plane had already been shot, it went on to the next

subroutine. We wanted to reduce the options from 1 to 12 depending on whether the end planes were shot off. After entering what we thought were the proper statements, the program still didn't seem to function as quickly as we thought it should. We modified the scoring subroutine to print the high and low search values in the space provided for score. By doing this, we were able to play the game and keep track of the variables at the same time.

Wrapping it up, we'll add the old adage "an ounce of prevention is worth a pound of cure". The best way to debug is to do it as you go along. Besides, it's more fun that way since you get to run the program sooner.

```
**************************
*     KAMAKAZE  RUN      *
*       V-PE331KB        *
*     BY T CASTLE        *
**************************
```

DESCRIPTION. Don't be deceived by the size of this program. It contains an the program begins running, the player is given a black screen with two rows of Kamakaze planes at the top. There are twelve planes in each row. The bottom of the screen has a green band with three white buildings to the left and a hovering, blue gunship to fend of the Kamakaze pilots. Superimposed on the green band is a white zero to the left. This number will be replaced later by the high score during each session of play. The white zero to the right is where the current score will be recorded as the game progresses. In the center of the screen there is a message instructing the player to "HIT ANY KEY".

After hitting a key, the lowest row of pilots (red) begin to drop, randomly, toward the ground. If they are permitted to drop to a level just above the blue gunship, they will begin a bombing run across the screen to the left and bomb the first building still standing. You control the movement of the blue gunship with the left and right arrow and fire using the "period". When enough of the red planes have started dropping, the yellow planes begin dropping. They will not come below the level of the highest red plane. Once they start their bomb run, you cannot shoot the plane down. If you lose all three buildings the game is over and the opening display is again put up. If you clear the board by shooting down all 24 planes (red and yellow) your buildings are rebuilt and 24 more planes are again placed at the top. With each successive board, the planes drop more quickly and the score for each kill is increased. You can pause at any time by hitting the "P". A "PAUSE" message is placed on the screen and you can begin again by hitting any key. The values of the red planes on levels 1, 2, and 3 are 5Ø, 1ØØ, 15Ø points respectively. The value of the yellow planes on levels 1, 2, and 3, are 15Ø, 225, and 3ØØ points respectively. Consider yourself fortunate if you get get to the fourth board and a score of 2Ø,ØØØ points or over.

NOTES. This program is layed out very much like the discussion of game programs in Chapter 2. The general sequence is found in lines 1ØØ-3ØØ and the main subroutines are as follows: initial variables lines 31Ø-62Ø; starting display lines 63Ø-97Ø; movement of row 2 and branching statement to bomb run (line 113Ø) lines 98Ø-127Ø; movement of row 1 and branching statement to bomb run (line 141Ø) lines 128Ø-171Ø; gunship movement and scoring routine lines 172Ø-224Ø. Additional subroutines for bomb attack, printing score, printing high score, and printing messages are found at 225Ø, 293Ø, 3Ø1Ø, and 3Ø9Ø. The variable LVL controls how many rows each plane drops on each movement. Scoring is based on the LVL value and is found in lines 2Ø6Ø and 215Ø.

This program is primarily made possible by setting up two arrays (line 33Ø, R1 and R2) which keep track of the current row location for each of the 24 planes which are dropping.

```
100 REM  ****************
110 REM  * KAMIKAZE RUN *
120 REM  ****************
130 REM  BY T CASTLE
140 REM  AMLIST V-PE331KB
150 REM
160 GOSUB 320
170 GOSUB 640
180 GOSUB 1730
190 IF EN=3 THEN 160
200 IF MX1+MX2=24 THEN 170
210 GOSUB 990
220 IF EN=3 THEN 160
230 IF MX1+MX2=24 THEN 170
240 GOSUB 1730
250 IF EN=3 THEN 160
260 IF MX1+MX2=24 THEN 170
270 GOSUB 1290
280 IF EN=3 THEN 160
290 IF MX1+MX2=24 THEN 170
300 GOTO 180
310 REM  SET START VALUES
320 CALL CLEAR
330 DIM R1(12),R2(12)
340 IF SCR>HSCR THEN 370 ELS
E 350
350 HSCR=HSCR
360 GOTO 380
370 HSCR=SCR
380 EN=0
390 SCR=0
400 LVL=2
410 CALL SCREEN(2)
420 CALL COLOR(3,16,13)
430 CALL COLOR(4,16,13)
440 CALL COLOR(5,16,1)
450 CALL COLOR(6,16,1)
460 CALL COLOR(7,16,1)
470 CALL COLOR(8,16,1)
480 CALL COLOR(13,11,1)
490 CALL COLOR(14,7,1)
500 CALL COLOR(15,13,1)
510 CALL COLOR(16,16,1)
520 CALL COLOR(12,5,1)
530 E1$="007E7E3C3C181800"
540 CALL CHAR(128,E1$)
550 CALL CHAR(136,E1$)
560 CALL CHAR(137,"1084200A8
0240000")
570 CALL CHAR(144,"FFFFFFFF
FFFFFFF")
580 CALL CHAR(152,"0F0909FFF
F9999FF")
590 CALL CHAR(120,"18187E7EF
FFF7E00")
600 CALL CHAR(153,"000018181
8180000")
610 CALL CHAR(154,"000000000
0006FFF")
620 RETURN
630 REM  START DISPLAY
640 LVL=LVL+1
650 FOR I=1 TO 12
660 R1(I)=2
670 R2(I)=1
680 NEXT I
690 MX1=0
700 MX2=0
710 EN=0
720 L1=1
730 L2=13
740 L3=1
750 L4=13
760 P1=16
770 FOR I=1 TO 22
780 CALL HCHAR(I,1,32,32)
790 NEXT I
800 FOR J=5 TO 27 STEP 2
810 CALL HCHAR(1,J,128)
820 CALL HCHAR(2,J+1,136)
830 NEXT J
840 CALL HCHAR(24,1,144,32)
850 GOSUB 2940
860 GOSUB 3020
870 CALL HCHAR(23,4,152)
880 CALL HCHAR(23,6,152)
890 CALL HCHAR(23,8,152)
900 CALL HCHAR(22,P1,120)
910 IF SCR>0 THEN 970
920 MSG$="1210HIT ANY KEY"
930 GOSUB 3100
940 CALL KEY(3,KY,ST)
950 IF ST=0 THEN 940
960 CALL HCHAR(12,10,32,12)
970 RETURN
980 REM  MOVES ROW 2
990 IF MX2=12 THEN 1270
1000 RANDOMIZE
```

```
1010 T1=0
1020 IF R2(L3)=25 THEN 1030
ELSE 1050
1030 L3=L3+1
1040 GOTO 1060
1050 L3=L3
1060 IF R2(L4-1)=25 THEN 107
0 ELSE 1090
1070 L4=L4-1
1080 GOTO 1100
1090 L4=L4
1100 M1=INT((L4-L3)*RND)+L3
1110 IF R2(M1)>=25 THEN 1270
1120 IF R2(M1)>21-LVL THEN 1
130 ELSE 1170
1130 GOSUB 1620
1140 IF EN=3 THEN 1270
1150 R2(M1)=25
1160 GOTO 1260
1170 FOR T=1 TO 12
1180 IF R1(T)<=R2(M1)+LVL TH
EN 1190 ELSE 1210
1190 T1=1
1200 T=12
1210 NEXT T
1220 IF T1=1 THEN 1270
1230 CALL SOUND(500,-4,0)
1240 CALL HCHAR(R2(M1)+LVL,(
M1*2)+3,128)
1250 CALL HCHAR(R2(M1),(M1*2
)+3,32)
1260 R2(M1)=R2(M1)+LVL
1270 RETURN
1280 REM  MOVES ROW 1
1290 IF MX1=12 THEN 1490
1300 IF R1(L1)=25 THEN 1310
ELSE 1330
1310 L1=L1+1
1320 GOTO 1340
1330 L1=L1
1340 IF R1(L2-1)=25 THEN 135
0 ELSE 1370
1350 L2=L2-1
1360 GOTO 1380
1370 L2=L2
1380 M1=INT((L2-L1)*RND)+L1
1390 IF R1(M1)>=25 THEN 1490
1400 IF R1(M1)>21-LVL THEN 1
410 ELSE 1450
1410 GOSUB 1510
1420 IF EN=3 THEN 1490
1430 R1(M1)=25
1440 GOTO 1490
1450 CALL SOUND(500,-4,0)
1460 CALL HCHAR(R1(M1)+LVL,(
M1*2)+4,136)
1470 CALL HCHAR(R1(M1),(M1*2
)+4,32)
1480 R1(M1)=R1(M1)+LVL
1490 RETURN
1500 REM    BOMB RUN 1
1510 CALL VCHAR(20-LVL,(M1*2
)+4,32,LVL+2)
1520 FOR I=(M1*2)+4 TO 8 STE
P -1
1530 CALL SOUND(200,-4,0)
1540 CALL HCHAR(21,I,136)
1550 CALL HCHAR(21,I,32)
1560 NEXT I
1570 MX1=MX1+1
1580 VL1=136
1590 GOSUB 2260
1600 RETURN
1610 REM  BOMB RUN 2
1620 CALL VCHAR(20-LVL,(M1*2
)+3,32,LVL+2)
1630 FOR I=(M1*2)+3 TO 8 STE
P -1
1640 CALL SOUND(200,-4,0)
1650 CALL HCHAR(21,I,128)
1660 CALL HCHAR(21,I,32)
1670 NEXT I
1680 MX2=MX2+1
1690 VL1=128
1700 GOSUB 2260
1710 RETURN
1720 REM  GUN MOVEMENT
1730 CALL KEY(3,KY,ST)
1740 IF ST=0 THEN 2240
1750 IF KY=80 THEN 1790
1760 IF KY=83 THEN 1860
1770 IF KY=68 THEN 1910
1780 IF KY=46 THEN 1960 ELSE
 2240
1790 MSG$="1213PAUSE"
1800 GOSUB 3100
1810 CALL KEY(3,KY,ST)
1820 IF ST=0 THEN 1810
1830 MSG$="1213     "
1840 GOSUB 3100
```

```
1850 GOTO 1730
1860 IF P1-1<5 THEN 2240
1870 P1=P1-1
1880 CALL HCHAR(22,P1+1,32)
1890 CALL HCHAR(22,P1,120)
1900 GOTO 2240
1910 IF P1+1>28 THEN 2240
1920 P1=P1+1
1930 CALL HCHAR(22,P1-1,32)
1940 CALL HCHAR(22,P1,120)
1950 GOTO 2240
1960 G1$=STR$((P1/2)-2)
1970 IF VAL(G1$)<1 THEN 1990
1980 IF LEN(G1$)<3 THEN 2080
1990 G2=VAL(G1$)+.5
2000 G1=R2(G2)
2010 IF G1=25 THEN 2020 ELSE
 2040
2020 G1=1
2030 GOTO 2070
2040 R2(G2)=25
2050 MX2=MX2+1
2060 SCR=SCR+((LVL-1)*75)
2070 GOTO 2160
2080 G2=VAL(G1$)
2090 G1=R1(G2)
2100 IF G1=25 THEN 2110 ELSE
 2130
2110 G1=1
2120 GOTO 2160
2130 R1(G2)=25
2140 MX1=MX1+1
2150 SCR=SCR+((LVL-2)*50)
2160 FOR G=21 TO G1 STEP -2
2170 CALL SOUND(50,-3,0)
2180 CALL HCHAR(G,P1,153)
2190 CALL HCHAR(G,P1,32)
2200 NEXT G
2210 CALL HCHAR(G1,P1,32)
2220 CALL SOUND(300,-5,0,120
,0)
2230 GOSUB 2940
2240 RETURN
2250 REM  BOMB ATTACK 1
2260 CALL GCHAR(23,8,BL1)
2270 CALL GCHAR(22,8,TST)
2280 IF TST=120 THEN 2290 EL
SE 2300
2290 EN=2
2300 IF BL1=152 THEN 2570
2310 FOR I=7 TO 6 STEP -1
2320 CALL SOUND(200,-4,0)
2330 CALL HCHAR(21,I,VL1)
2340 CALL HCHAR(21,I,32)
2350 NEXT I
2360 CALL GCHAR(23,6,BL1)
2370 CALL GCHAR(22,4,TST)
2380 IF TST=120 THEN 2390 EL
SE 2400
2390 EN=2
2400 IF BL1=152 THEN 2670
2410 FOR I=5 TO 4 STEP -1
2420 CALL SOUND(200,-4,0)
2430 CALL HCHAR(21,I,VL1)
2440 CALL HCHAR(21,I,32)
2450 NEXT I
2460 CALL GCHAR(23,4,BL1)
2470 CALL GCHAR(22,4,TST)
2480 IF TST=120 THEN 2490 EL
SE 2500
2490 EN=2
2500 IF BL1=152 THEN 2770
2510 FOR I=3 TO 1 STEP -1
2520 CALL SOUND(200,-4,0)
2530 CALL HCHAR(21,I,VL1)
2540 CALL HCHAR(21,I,32)
2550 NEXT I
2560 GOTO 2920
2570 CALL HCHAR(22,8,153)
2580 CALL HCHAR(22,8,32)
2590 CALL HCHAR(23,8,153)
2600 CALL HCHAR(23,8,137)
2610 CALL SOUND(200,-5,0,170
,0)
2620 CALL SOUND(450,-5,0,120
,0)
2630 CALL HCHAR(23,8,154)
2640 EN=EN+1
2650 J=7
2660 GOTO 2870
2670 CALL HCHAR(22,6,153)
2680 CALL HCHAR(22,6,32)
2690 CALL HCHAR(23,6,153)
2700 CALL HCHAR(23,6,137)
2710 CALL SOUND(200,-5,0,170
,0)
2720 CALL SOUND(450,-5,0,120
,0)
2730 CALL HCHAR(23,6,154)
2740 EN=EN+1
```

```
2750 J=5
2760 GOTO 2870
2770 CALL HCHAR(22,4,153)
2780 CALL HCHAR(22,4,32)
2790 CALL HCHAR(23,4,153)
2800 CALL HCHAR(23,4,137)
2810 CALL SOUND(200,-5,0,170
,0)
2820 CALL SOUND(450,-5,0,120
,0)
2830 CALL HCHAR(23,4,154)
2840 EN=EN+1
2850 J=3
2860 GOTO 2870
2870 FOR I=J TO 1 STEP -1
2880 CALL SOUND(200,-4,0)
2890 CALL HCHAR(21,I,VL1)
2900 CALL HCHAR(21,I,32)
2910 NEXT I
2920 RETURN
2930 REM  PRINT SCORE
2940 MS$=STR$(SCR)
2950 L=LEN(MS$)
2960 FOR I=1 TO L
2970 MS=ASC(SEG$(MS$,I,1))
2980 CALL HCHAR(24,20+I,MS)
2990 NEXT I
3000 RETURN
3010 REM  PRINT HI SCORE
3020 MS$=STR$(HSCR)
3030 L=LEN(MS$)
3040 FOR I=1 TO L
3050 MS=ASC(SEG$(MS$,I,1))
3060 CALL HCHAR(24,8+I,MS)
3070 NEXT I
3080 RETURN
3090 REM  PRINT ANY MSG
3100 MSR=VAL(SEG$(MSG$,1,2))
3110 MSC=VAL(SEG$(MSG$,3,2))
3120 L=LEN(MSG$)-4
3130 MS$=SEG$(MSG$,5,L+4)
3140 FOR I=1 TO L
3150 MS=ASC(SEG$(MS$,I,1))
3160 CALL HCHAR(MSR,MSC+I,MS
)
3170 NEXT I
3180 RETURN
```

HAPPY COMPUTING!

CHAPTER FOUR

Developing Graphics

GENERAL. For games and educational programs an understanding of the graphics capability of the 99/4A is essential. For functional type programs like checkbook management, mailing lists, etc., it may not be essential; however, it can make a dull task more interesting. Creating and using graphics in console basic is a four part process involving: defining of the character; assigning the defined character a specific character number; selecting the color combinations for the character number assigned; and finally, printing the character to a specific point on the screen. To explain the thought process involved and the relationship between each step, we're going to use the Patience Please program (card game), as an example. With the fundamentals clearly in mind, we'll then cover: movement of characters, blinking lights, magnification, and some other specialized techniques.

Prior Planning. Creating and coding characters can be a very time consuming process; therefore, before you spend a lot of time creating them, make sure that what you're thinking about doing can, in fact, be accomplished on the 99/4A. Television, arcade games, and programs in module form, often can provide you with "thrilling" ideas that you would like to incorporate into your own programs. Some of these ideas are transferable and some are not. We're not suggesting that the capabilities are small, in fact they are quite great; however, we are saying that you should be aware of some basic limitations and that you should be realistic in your expectations. Here are some popular ideas. As you read each one, consider whether it can be done with console basic.

1. A map of the US with an outline of each state.
2. A map of the Southeast with each state oulined.
3. A plane in the air, with the background moving.
4. A dancing bear, as big as the screen.
5. A race car at bottom of screen with road & background coming toward you.

By the end of this chapter you should be able to figure out whether these things are possible. For any particular application, the main questions which you need to ask yourself are these:

1. How many characters will I need to define?
2. How many color combinations do I need?
3. How many characters am I moving (if any) at one time?

Following is a quick review of the character coding system and then we'll give you a practical example of how to develop a graphics program.

Shorthand Coding. There are several charts available in the user's manual that came with your computer showing the shorthand codes for defining

characters. When coding characters you may refer to these or the chart below which is organized in a slightly different manner.

```
|  |  |  |  | - Ø     |XX|XX|XX|  | - E
|XX|XX|XX|XX| - F     |  |XX|XX|XX| - 7

|XX|  |  |  | - 8     |XX|  |XX|  | - A
|  |XX|  |  | - 4     |  |XX|  |XX| - 5
|  |  |XX|  | - 2
|  |  |  |XX| - 1     |XX|  |  |XX| - 9

|XX|XX|  |  | - C     |XX|XX|  |XX| - D
|  |XX|XX|  | - 6     |XX|  |XX|XX| - B
|  |  |XX|XX| - 3
```

Each of these codes, Ø - 9 and A - F will represent one digit of the sixteen digit code required to define a single screen character. The order in which they are listed in the sixteen digit code is as follows:

1	2
3	4
5	6
7	8
9	1Ø
11	12
13	14
15	16

Each character will ultimately be made up of 64 individual blocks with the light turned on or off. Each of the shorthand codes represents four of these blocks, and a combination of sixteen shorthand codes represents the entire character. With these codes, any shape, design, or character, which can be created on a 8 X 8 grid, can be defined. If these points are not clear, review your user's manuals and the above until you have a thorough understanding of this principle, since it is a prerequisite to the following discussions.

Defining Characters. There is a character code generator program in the user's manual, and others commercially available, which you may want to use when you start out coding characters. After you've coded a dozen or more characters, we think you'll agree that the system is not that complex. At that point, you'll probably find that coding from a rough sketch is faster than loading a program and using a code generator for each character. If you intend to do a lot of work with graphics we suggest that you buy some specialized graph paper.

As a standard, we suggest a sheet of paper 22" X 17", broken down into 1" squares (22 across by 17 high), with each 1" square further divided into 8 X 8 blocks. This gives you 64 smaller blocks, in a 1" X 1" larger block. You'll be able to find this at most retail stores that sell blueprints or supplies for draftsmen and architects. By putting two sheets like this side by side you can represent an entire screen display of 22 X 34 characters and still have individual blocks large enough to work with and write on. To design a plane, boat, rocket ship, or whatever, simply sketch the outline within the 1" block (for a 1 X 1 character), and shade the inside of the outline. If you use the type of graph paper mentioned above, it's quite easy to simply compare each of the 4 smaller block areas with the shorthand chart to come up with the sixteen digit code.

If you're not good at free hand artwork and you want to create something such as a bear or monkey, try finding a picture in a magazine or children's coloring book to use for a guide. Place a piece of carbon paper

over the graph paper, lay your sample on top, and then trace around the outline.

Patience Please. As a practical example of a graphics program, we'd like to dicuss how the Patience Please solitaire card game was developed. We started with the idea that we wanted the cards to look "real", not just number designations such as: 9-D for nine of diamonds; 8-S for eight of spades. We wanted the actual shapes for diamonds, clubs, hearts and spades. Further, we wanted the shapes red and black, and we wanted the numbers to be red and black also. We wanted them on white cards and we wanted it to look like a real card game laying on a table. With these thoughts in mind, before we ever started coding, we had to decide whether it was possible. To approach a problem like this, think about the total screen layout first, then work your way down to the individual characters.

Laying this out on a table we knew right away that we needed seven cards across the screen. If we consider that we have only 28 columns across the screen to work with, this meant that a card could only be 3 characters wide, plus a space between cards (7X4=28). Since a card is rectangular in shape, we figured it would have to be at least 5 characters long (from top to bottom) to look fairly real. On a solataire layout you have finished stacks at the top where the upper card is fully exposed. Below that you have stacks with overlapping cards, where only the last card is fully exposed. With 24 rows to work with we tried to determine if we could get enough cards on the screen. The finished stacks would consume 5 plus a blank row for separation. That left us rows 6 through 24. Row 2Ø through 24 had to be reserved for the last card in each stack. That left rows 7 through 19 that could be used for overlapping cards. Including the last card we could get a total of 14 cards in a stack. Although this game sometimes can require more than 14 cards in a stack, it doesn't occur often and we figured it was a limitation we could live with. The general layout looked alright, except for one thing. If we used the last line (and we really needed it for the stacks), where would we have our input statements indicating what move the person was making? Since the first six rows would only have four stacks, we figured if they were to the right side, we would have plenty of room for input in the upper left hand corner of the screen.

This is where a graphics program begins. The general layout has already dictated the size of each card and therefore, the size of the diamond, club, spade, and heart. We're only going to have 3 characters, from left to right, to designate any card. If one or two of these are used for the characters 1-1Ø and A, J, Q, and K, then only one can be used for the suit. The next thing we had to do was determine what characters needed to be defined and what colors we would use.

CALL SCREEN/COLOR. Unless you define it otherwise, the standard color for the screen is "Light Green" or color code 4. The standard color for the characters that are printed is black on transparent (which means they show up black on what ever color screen you have). To change these you need to use a CALL SCREEN or CALL COLOR command. We definitely wanted a dark green background like a poker table so

we were going to need a CALL SCREEN(13). Our input questions could be black on clear (the standard colors) so no change was necessary there. We needed two characters (diamond and heart) created that would be red on white (the color of the card), and two others (spade and club) that would be black on white. Showing the numbers on the card is where we ran into a problem.

If you recall the charts in your manuals, all letters, symbols, signs, etc. are assigned a unique code number, such as: 65 for a capital "A"; 32 for a "space"; and 52 for the number "4". A certain number of character codes, from 128 to 159 are not used at all initially. All characters are grouped into sets, each set containing 8 characters. You can't just change the color of one character, you must change an entire set of 8 at one time. Looking at the chart, the numerals Ø-9 are found in sets 3 and 4. To get an A (Ace), J (Jack), Q (Queen), and K (King), meant using sets 6 and 7 as well. If these were used for input (black on green), could they also appear on the screen as red on white or black on white? Of course, the answer is "they can't". Whenever you do a CALL COLOR statement, changing the colors in a particular set, all characters in that set already on the screen immediately change to the new color. This meant that using the standard numeral codes, 48-57, they could only appear as one color. We realized at this point that we had to redefine at least 13 more characters in red on white and 13 more black on white. Counting the suits, we needed 15 characters red on white and 15 black on white. Since each set of characters contains eight codes, we needed two sets for each color combination. Therefore, we would need the following CALL COLOR statements:

```
>1Ø1Ø CALL COLOR(13,7,16)
>1Ø2Ø CALL COLOR(14,7,16)
>1Ø3Ø CALL COLOR(15,2,16)
>1Ø4Ø CALL COLOR(16,2,16)
```

To this point you'll note that we haven't really defined any characters, we've just determined what had to be coded. Sometimes the calculation will work out, and other times it won't. In the Building Blocks program we originally wanted six different shapes, each available in 3 sizes and 4 colors. The finished program allowed for only 3 shapes, available in 3 sizes and 4 colors each. Our original calculations showed us that it just wasn't possible. If you use all of the characters from 32 to 159 you have 128 different characters that can be redefined. Can you code a map of the US, with every state outlined, using only 128 different shapes? You'll save time in the long run by giving some serious thought to the overall layout and requirements of the program before you begin actually developing the 16 digit codes. In our case, since it was possible, that was our next step.

<u>Coding Characters.</u> We started our character coding by sketching a "Heart", "Club", "Diamond", and a "Spade", into each of four 1" grids. After this was done we carefully shaded in all of the little inner blocks inside of the item we wanted to create. Next, we referred to our shorthand chart and wrote down the sixteen digit code representing the 64 smaller blocks. Lastly, we assigned each of these codes a string variable name. Following is what we came up with:

```
HRT$="ØØC6EEFE7C7C381Ø"
DMD$="ØØ1Ø387CFE7C381Ø"
SPD$="ØØ1Ø387CFEFED638"
CLB$="ØØ3838FEFEFE1Ø7C"
```

That took care of the odd shapes, now all we had to do was define the numerals and the A, J, Q, K. With each of their 16 digit codes, we could create a set of numbers in red on white with characters 128 to 141 and black on white with characters 144 to 157. Fortunately, it's not necessary to sketch out each of the numbers and figure out their codes. In extended basic there is a command that will return the 16 digit pattern identifier for any predefined character. For your convenience, we've listed these on a chart at the end of this chapter. Using the code for the numerals, with a slight change that places a line across the top, we had all of our character definitions.

Assigning Numbers. We already decided which sets were what color, so the next thing we had to do was assign the above codes to a particular ASC number within the appropriate set. Following are three methods of doing this. We haven't listed all of the codes, just those for the suit.

Method 1.

```
>1ØØØ CALL CHAR(142,"ØØC6EEFE
 7C7C381Ø")
>1Ø1Ø CALL CHAR(143,"ØØ1Ø387C
 FE7C381Ø")
>1Ø2Ø CALL CHAR(158,"ØØ1Ø387C
 FEFED638")
>1Ø3Ø CALL CHAR(159,"ØØ3838FE
 FEFE1Ø7C")
```

Method 2.

```
>1ØØØ HRT$="ØØC6EEFE7C7C381Ø"
>1Ø1Ø DMD$="ØØ1Ø387CFE7C381Ø"
>1Ø2Ø SPD$="ØØ1Ø387CFEFED638"
>1Ø3Ø CLB$="ØØ3838FEFEFE1Ø7C"
>1Ø4Ø CALL CHAR(142,HRT$)
>1Ø5Ø CALL CHAR(143,DMD$)
>1Ø6Ø CALL CHAR(158,SPD$)
>1Ø7Ø CALL CHAR(159,CLB$)
```

Method 3.

```
>1ØØØ DATA ØØC6EEFE7C7C381Ø,Ø
 Ø1Ø387CFE7C381Ø,ØØ1Ø387CFEFE
 D638,ØØ3838FEFEFE1Ø7C
>1Ø1Ø RESTORE 1ØØØ
>1Ø2Ø FOR I=142 TO 143
>1Ø3Ø READ A$
>1Ø4Ø CALL CHAR(I,A$)
>1Ø5Ø NEXT I
>1Ø6Ø FOR I=158 TO 159
>1Ø7Ø READ A$
>1Ø8Ø CALL CHAR(I,A$)
>1Ø9Ø NEXT I
```

Which method is the best? If your goal is to set your characters in the fewest number of lines, for four characters, it looks like the first method is the shortest. The real answer is, "It depends on how many characters need to be created and what those characters are". In the program we're developing we will ultimately have 3Ø different characters to be assigned and we're going to use almost all of the codes from 128 to 159. Using method 1 it would take 3Ø lines; method 2 would take 6Ø lines; and method 3 would take about 13 lines. In this case, method 3 is the shortest because the FOR-NEXT loops can assign codes sequentially while READing from a DATA line. For ease in typing, we didn't condense it as much as we could have in our finished program, but the assignments are made in lines 4ØØ-59Ø (2Ø lines). In the "Building Blocks" program, lines 4ØØ to 57Ø, we defined a total of 6Ø characters in just 17 lines. In that case, we had fifteen different character definitions, each

of which had to be assigned four different code numbers. Generally, when you're developing a program, method 2 will be the easiest, since you can use descriptive variable names to define your characters. As you continue your program, if you find you need to change some definitions, they're easier to find and correct. When the program is complete and running, you can go back and rewrite your definition section to the most compact method.

In a moment we're going to get into the exciting part of graphics, i.e. moving characters around, and at this point the "Patience Please" program isn't going to be all that exciting. Since it is the basis for any card game, we do want to point out a couple of final things about this program. First, the ten was not defined as two characters using the predefined codes at the end of the chapter. We designed a separate character which had both the 1 and the Ø within the same character. Second, the creation of the 52 card deck is found in the subroutine beginning at 33Ø and ending at 73Ø. Almost all of these statements would be required for Poker, Blackjack, etc. The exceptions are the DIM statements for TBU$, TBD$, ELG$ AND CODE$. The deck that is created is called DECK$(52). In lines 95Ø to 11ØØ you'll find a shuffle routine that shuffles the array called DECK$. The variable "Z" in line 97Ø is a flag to indicate whether the deck was previously shuffled. The first time a program goes through this routine it makes 1Ø4 swaps; thereafter, it make 52 swaps on each pass. You could adjust these up or down to give you more or less shuffle. Where you place the cards on the screen would of course depend on the game your creating. If you don't want to code in the entire card game program, the following small program prints the seven card stacks with the diamonds, clubs, spades, and hearts in the appropriate colors on each card.

```
>1ØØ CALL CLEAR
>11Ø GOSUB 1ØØØ
>12Ø GOSUB 2ØØØ
>13Ø GOSUB 3ØØØ
>14Ø GOTO 14Ø
>1ØØØ CALL CHAR(128,"ØØC6EEFE
 7C7C381Ø")
>1Ø1Ø CALL CHAR(129,"ØØ1Ø387C
 FE7C381Ø")
>1Ø2Ø CALL CHAR(136,"ØØ1Ø387C
 FEFED638")
>1Ø3Ø CALL CHAR(137,"ØØ3838FE
 FEFE1Ø7C")
>1Ø4Ø CALL CHAR(143,"Ø")
>1Ø5Ø RETURN
>2ØØØ CALL SCREEN(13)
>2Ø1Ø CALL COLOR(13,7,16)
>2Ø2Ø CALL COLOR(14,2,16)
>2Ø3Ø RETURN
>3ØØØ FOR I=2 TO 26 STEP 4
>3Ø1Ø CALL VCHAR(1Ø,I,143,5)
>3Ø2Ø CALL VCHAR(1Ø,I+1,143,5
 )
>3Ø3Ø CALL VCHAR(1Ø,I+2,143,5
 )
>3Ø4Ø J=J+1
>3Ø5Ø IF J<5 THEN 3Ø7Ø
>3Ø6Ø J=1
>3Ø7Ø ON J GOTO 3Ø8Ø,31ØØ,312
 Ø,314Ø
>3Ø8Ø A=128
>3Ø9Ø GOTO 315Ø
>31ØØ A=129
>311Ø GOTO 315Ø
>312Ø A=136
>313Ø GOTO 315Ø
>314Ø A=137
>315Ø CALL HCHAR(1Ø,I,A)
>316Ø NEXT I
>317Ø RETURN
```

Creating Movement. Aside from the PRINT statements, the only way to put something on the screen is through use of the CALL HCHAR or CALL VCHAR command. Their use is well documented in the reference manuals and simply involves specifying a row, column, and character number. If you want more than one on a row or column you can add a fourth variable for repetitions. If you are in doubt as to the basic command, now is the time to read up on it. The following sample program is devoted more to the methods of moving the character, using the CALL KEY and CALL JOYST command, rather than to the CALL HCHAR command itself.

In the one sample program below we have given you all of the basic ingredients for a typical "move 'em and shoot 'em" type game. It starts by clearing the screen and changing the screen color to black. Next, a red "target" block will appear at a random point on the screen. A white block, your "ship", appears in the upper left hand corner. You control movement of your ship with either of the joysticks or the keyboard. The keyboard only shows up/down, or left/right arrows. So that you can move diagonally, as you can with joysticks, we've also mapped the keyboard to accept the "R", "W", "Z", and "C" keys for diagonal movement. You can "fire" by hitting a "period" on the keyboard or the firing button on either joystick. Manipulate the white block until it contacts the red block. When you make contact you'll get an "explosion" and the game will restart.

The "explosion" and "firing" mentioned above are just sounds indicating where this would occur. You can add your own embellishments to this and create your own custom designed game. We're going to spend a little time discussing this so we suggest you enter it at this point.

```
1ØØ CALL CLEAR
11Ø RANDOMIZE
12Ø CALL CHAR(128,"ØØ7E7E7E7
E7E7EØØ")
13Ø CALL CHAR(136,"FFFFFFFFF
FFFFFFF")
14Ø CALL SCREEN(2)
15Ø CALL COLOR(13,12,1)
16Ø CALL COLOR(14,7,1)
17Ø C=3
18Ø R=3
19Ø CH=128
2ØØ GOSUB 71Ø
21Ø GOSUB 83Ø
22Ø REM  CALL KEYS
23Ø CALL KEY(3,KY(3),ST(3))
24Ø CALL KEY(2,KY(2),ST(2))
25Ø CALL KEY(1,KY(1),ST(1))
26Ø CALL JOYST(1,KY(4),ST(4)
)
27Ø CALL JOYST(2,KY(5),ST(5)
)
28Ø IF (ST(1)=Ø)*(ST(2)=Ø)*(
ST(3)=Ø)*(ST(4)=Ø)*(ST(5)=Ø)
*(KY(4)=Ø)*(KY(5)=Ø)THEN 23Ø
29Ø IF KY(3)>=Ø THEN 38Ø
3ØØ IF (ST(1)=Ø)*(ST(2)=Ø)TH
EN 33Ø
31Ø GOSUB 67Ø
32Ø GOTO 23Ø
33Ø IF (KY(4)=Ø)*(KY(5)=Ø)*(
ST(4)=Ø)*(ST(5)=Ø)THEN 38Ø
34Ø R=ROW+(-.25*(ST(4)+ST(5)
))
35Ø C=COL+(.25*(KY(4)+KY(5))
)
36Ø GOSUB 71Ø
37Ø IF X=136 THEN 1ØØ ELSE 2
3Ø
38Ø IF KY(3)=46 THEN 31Ø
39Ø IF KY(3)<>67 THEN 43Ø
4ØØ R=ROW+1
41Ø C=COL+1
42Ø GOTO 36Ø
43Ø IF KY(3)<>68 THEN 46Ø
```

```
440 C=C+1
450 GOTO 360
460 IF KY(3)<>69 THEN 490
470 R=R-1
480 GOTO 360
490 IF KY(3)<>82 THEN 530
500 R=R-1
510 C=C+1
520 GOTO 360
530 IF KY(3)<>83 THEN 560
540 C=C-1
550 GOTO 360
560 IF KY(3)<>87 THEN 600
570 R=R-1
580 C=C-1
590 GOTO 360
600 IF KY(3)<>88 THEN 630
610 R=R+1
620 GOTO 360
630 IF KY(3)<>90 THEN 230
640 R=R+1
650 C=C-1
660 GOTO 360
670 REM  SHOOT
680 IF (KY(1)=18)+(KY(2)=18)
+(KY(3)=46)THEN 690 ELSE 700
690 CALL SOUND(100,260,0)
700 RETURN
710 REM  MOVE CHARACTER
720 IF (R<2)+(R>23)+(C<2)+(C
>31)THEN 800
730 ROW=R
740 COL=C
750 CALL GCHAR(ROW,COL,X)
760 CALL HCHAR(ROW,COL,CH)
770 IF X<>136 THEN 800
780 GOSUB 880
790 GOTO 820
800 R=ROW
810 C=COL
820 RETURN
830 REM  PLACE TARGET
840 RT=INT(16*RND)+5
850 CT=INT(16*RND)+5
860 CALL HCHAR(RT,CT,136)
870 RETURN
880 REM  EXPLOSION
890 CALL SOUND(1000,-3,0)
900 RETURN
```

Program Layout. This program is broken down into several main parts which we will discuss. The initial statements which include: the randomize statement, character definitions, CALL SCREEN, CALL COLOR, and starting point for your ship, are set in lines 100-190. Lines 200-210 use subroutines near the end of the program to print your ship initially and randomly place the target. From 220-270 we perform all of the CALL KEY's and CALL JOYST's required of the program. Lines 340-370 change values of ROW and COL based on joystick movement. Lines 390-660 change ROW and COL based on keyboard entries. IF statements throughout the program determine if you "fired" and recycle the program when a "hit" occurs. Separate subroutines are provided to MOVE CHARACTER, PLACE TARGET, to simulate EXPLOSION, and SHOOT.

This program allows input from all sources. It should be noted that this isn't always necessary, nor desirable. Allowing for all choices will slow down the response rate since the computer has more checking to do for every pass through the call key. If you are really concerned with the fastest possible speed, use only one of these methods, or make it an option at the beginning of the program. To see what kind of difference it will make, run the program as written, then add the following temporary line:

```
>235 GOTO 280
```

With this line added the computer will only be looking for keyboard input. With an option at the beginning you could isolate the minimum number of statements required. Let's look at exactly what's required for each option.

Keyboard Movement. All that's required to generate movement or indicate a firing condition from the keyboard is one CALL KEY statement. We almost always use keyboard number 3, since it does not distinguish between upper and lower case letters. The alpha keys are always returned with their upper case ASC value. The only statement normally required for moving an item from the keyboard is:

```
>100 CALL KEY(3,KY,ST)
```

A series of IF statements would follow this entry. If status (ST) was 0 you would simply go back to the CALL KEY again, looking for a response. If it wasn't 0, the program would check for the ASC value of the key touched (KY). Depending on which key was touched, you would adjust the ROW and COL, or FIRE. If it was anything but an acceptable key, you send it back to the CALL KEY statement. These comparisons are made in lines 280, 290, 380-670 of this program. This program allows for diagonal movement. To allow for just horizontal or vertical movement, you can remove the references to character numbers 67, 82, 87, and 90, from the program.

Notice in the attached program that we use temporary variables for R and C for our adjustments. After the adjustment is made it goes to the print subroutine in lines 710-820. Before a permanent change is made in the value of ROW and COL we make sure that the adjusted value is within the range of acceptable numbers. If not, the values remain the same, R and C are returned to their pre-adjustment values, and the program RETURNs to the CALL KEY.

Joystick. In order to determine what is happening with either joystick, two commands are required per controller. The first command is a CALL KEY which is needed to register use of the "Fire" button. In the attached program, determination of a firing condition takes place in lines 240 & 250. The first number within parenthesis indicates which controller it's looking at (#1 or #2). To see the relationship between the values of KY (key) statements and ST (status) statements, remove line 140 from the previous program and add the following:

```
>275 PRINT KY(1);ST(1);TAB(10
 );KY(2);ST(2);TAB(20);KY(3);
 ST(3)
```

The way this program is set up we're actually looking at a "Split Keyboard Scan" and the "Standard Keyboard Scan". Refer to your charts in the user's reference manual and compare the values of KY that are returned when you use the keyboard. With the above test routine operating, hit the firing buttons on the joysticks. Each returns a value of 18 for KY, which is the same as a "Q" on the left side of the split scan keyboard or "Y" on the right side.

The second command necessary is a CALL JOYST. The format for this calls for a designation of either key unit 1 or key unit 2. Replace line 275 above with the following and move the two joysticks through their various positions:

```
>275 PRINT KY(4);ST(4);TAB(10
 );KY(5);ST(5)
```

Movement of the joystick will return either positive or negative values of 4 for either one or both of the variables in the CALL JOYST statement. The first variable (KY(4 or 5) in our

program) is the left or right movement. The second variable ST(4 or 5) is the up and down movement. To convert this to an adjustment for the R and C values we use the statements in 34Ø and 35Ø. By multiplying the value of KY by .25 we convert it to a value of "1". A -.25 multiplier is needed for row (ST) because row numbers increase from top to bottom.

Contacting Target. To determine whether contact is made between a moving character and a target, we use a CALL GCHAR just prior to the statement which will move the character. This is found in line 75Ø of this program. We use the ROW and COL that we are going to print to in the CALL GCHAR statement and it returns the ASC value of the character at that position as X. Our target has a value of 136. We test for that condition in 77Ø and go to an "explosion" if it's found. Notice that on RETURN from that subroutine, in line 37Ø, we again test for the value of X. If it has found it, the program is restarted.

Modifications. This is a good test program and a good base program for games you might want to create. We've left a "trace" when the white ship is moved. Remove the test line above and put back the CALL SCREEN(2), then enter the following lines to play the game without the trace:

```
>175 COL=3
>185 ROW=3
>725 CALL HCHAR(ROW,COL,32)
```

The CALL GCHAR command is used in line 75Ø of this program to check for contact. If you replace this with a REM statement, you'll notice that the white block moves considerably faster. GCHAR's do take time. An alternate way of checking for this is to compare the value of ROW and COL, which we are printing to, with the value of RT and CT, which is where we placed the target. This method is used in the Kamakaze Run program which stores locations in arrays.

Another place to experiment with this program is to use a variable such as SP (for speed) in place of the "1" which is added or subtracted from the R & C values in the adjustments. If you used SP=2 and changed the .25 multiplier in lines 34Ø and 35Ø to .5, the white block will move twice as fast. This certainly gives the illusion of far greater speed. Doing this will also require a change in the method of detecting contact. It would be possible, if it were moving two spaces at a time, that it might never hit the exact same spot as the target.

Practice with this program by redefining the characters to something more thrilling than two "blocks"; add greater sound effects for the explosion and firing routines; add bullets that shoot from the ROW and COL position in four directions when you hit the "Fire" button. You might surprise yourself with your own creative ability. Now let's move on to some other graphics techniques.

Blinking Lights. In the "Monkey Business" program at the end of Chapter 5, we are looking for a one digit response from the student and the response always appears at the same point on the screen. In order to highlight this spot and bring it to the student's attention we built a a 3 X 3 square around the spot where the answer is to appear. This is done as part of the START DISPLAY routine from 55Ø-69Ø. To make the block "blink" on and off we added a CALL COLOR

statement to each side of a CALL KEY statement in lines 3550-3580. Each time the program went through the CALL KEY, the block was turned on, so it appeared Red on Dark Green just prior to the CALL KEY, and then it was turned off, appearing Clear on Dark Green just after the CALL KEY. Even if we used CALL HCHAR's and CALL VCHAR's with added repetitions, it would always take four commands to create a block and four to erase it. If you reserve certain spaces on the screen for blocks, error messages, or other objects, and if you design your program so that you never write over these spaces during the running of the program, the CALL COLOR command can be a quick method of "popping" information to the screen all at one time, instead of character by character.

Magnified Characters. Another use for the character table at the end of this chapter is for the creation of "oversize" letters or "Titles". To do this in console basic it takes a rather extensive subroutine which accepts the 16 digit code for any letter, it analyzes it, and then it creates four 16 digit codes which go together to print the same letter again, except it appears two spaces high by two spaces wide. The result is a far more attractive "Title" than you would normally get using HCHAR's and VCHAR's.

Possible uses for this include: expanding the size of letters, such as an A or B, for use in an educational program for children; a title screen such as found in the "Happy Birthday" program in Chapter 5; or increasing the size of any character to make it appear to be coming toward you. The subroutine we're about to describe: first, accepts the 16 digit character code; second, it analyzes the code; and third, it returns the four character codes which, when printed in the proper order, give you the orginal character two spaces high by two spaces wide. Again, we're going to ask you to put in the following sample program so that you'll understand the principle. There are corresponding subroutines at lines 830, 1030, and 2690 of the "Birthday" program.

```
>100 CALL CLEAR
>110 GOSUB 1000
>120 INPUT "CODE       ":A$
>130 INPUT "START      ":A
>140 INPUT "ROW,COL ":ROW,COL
>150 CALL CLEAR
>160 GOSUB 2000
>170 GOSUB 3000
>180 GOTO 180
>1000 DIM BG$(16)
>1010 DATA 0000,0303,0C0C,0F0
 F,3030,3333,3C3C
>1020 DATA 3F3F,C0C0,C3C3,CCC
 C,CFCF,F0F0,F3F3
>1030 DATA FCFC,FFFF
>1040 RESTORE 1010
>1050 FOR I=1 TO 16
>1060 READ BG$(I)
>1070 NEXT I
>1080 RETURN
>2000 FOR BG=1 TO 16
>2010 BG1$=SEG$(A$,BG,1)
>2020 BG1=ASC(BG1$)
>2030 IF BG1<65 THEN 2060
>2040 BG1=BG1-54
>2050 GOTO 2070
>2060 BG1=BG1-47
>2070 BG2=BG2+1
>2080 IF BG>8 THEN 2110
>2090 B$(BG2)=B$(BG2)&BG$(BG1
 )
>2100 GOTO 2120
>2110 B$(BG2+2)=B$(BG2+2)&BG$
 (BG1)
>2120 IF BG2<2 THEN 2140
>2130 BG2=0
```

```
>2140 NEXT BG
>2150 FOR BG=0 TO 3
>2160 CALL CHAR((A+BG),B$(BG+
 1))
>2170 NEXT BG
>2180 RETURN
>3000 CALL HCHAR(ROW,COL,A)
>3010 CALL HCHAR(ROW,COL+1,A+
 1)
>3020 CALL HCHAR(ROW+1,COL,A+
 2)
>3030 CALL HCHAR(ROW+1,COL+1,
 A+3)
>3040 RETURN
```

After entering this sample, type RUN and wait for the input statements. When it asks for "CODE", enter any sixteen digit character code. We suggest you try one of the codes for the predefined characters listed on the chart at the end of this chapter, since you'll know what that character looks like in normal size. When it asks for "START", give it the first character number you want to redefine. Since 128 is the first undefined character, we suggest you try this first. The program will actually redefine 128, 129, 130, and 131. When it asks for "ROW,COL ", enter two numbers such as 10,10. These will represent the row and column where the upper left hand portion of the oversize letter will appear. After you enter the ROW and COL and hit the ENTER key, the screen will clear and the oversize letter will appear at the specified position. To try a different letter, simply do a FCTN-4 and RUN it again. Now let's go through a brief explanation of what's happened.

Before we ever received any input information, we sent the program through a beginning subroutine and set up an array called BG$. This array has 16 elements, each 4 characters long, taken from the data statements in lines 1010-1030. Look at the chart below and we'll try to make it clear just what these are used for.

```
CODE FOR "A" - 003844447C444444
SEGMENT      - 1212121234343434
```

Without going into every loop, here's what the subroutine beginning at 2000 does. It starts with a sixteen digit code, such as that shown for the "A" above. It needs to create four separate sixteen digit codes representing the four characters required to print the oversize character. Remember that there are sixteen options in the normal shorthand code (0-9 and A-F). These correspond to the sixteen, four digit codes we have set up under the array BG$. If we take all of the codes above the SEGMENT marked number 1, we have 0,3,4,4. A "0" is the first possible shorthand code, so we'll get the first four digit BG$ code, or BG$(1), which equals "0000". Three is the fourth possible shorthand code, so we'll get the fourth four digit BG$ code, or BG$(4), which equals "0F0F". The code for the four is BG$(5), or "3030", and we have two of these. Adding these together as a string we get "00000F0F3030". This code represents the code for the upper left hand portion of the oversize letter. The process is then repeated for segment 2, 3, and 4. After this is complete, in lines 2150 to 2170, we use these codes to redefine four characters beginning with your starting value (A). Finally, it's a simple matter to build a print routine, such as that in 3000-3040 which prints the four characters to the screen in the appropriate positions.

In practical use, you may have six, seven or more letters to be defined. You'll have to write some other subroutines which automatically keep feeding sixteen digit strings to GOSUB 2000. You'll also have to keep increasing the value of "A" each time through the loop so that, after characters 128-131 are used, it then begins with 132 through 135. Further, going to the print routine will require that you keep increasing the value of COL by at least 3 to print the letters side by side. If you study the referenced subroutines in the "Birthday" program, you'll see that we incorporated some of these loops right in with the main subroutines. CAUTION - These eat up a lot of characters. In the "Birthday" program we had 9 different letters to print so it required 36 characters or 5 sets. We also needed 4 sets for the cake and candles, so we had to begin redefining with character 88.

As we close this chapter, we can't help mentioning some of the advantages of this routine for those of you who have Extended Basic. If you have this, you can see how easy it would be to develop your sixteen digit code for MAGNIFY by simply doing a CALL CHARPAT and then feeding that value to the above routines.

CHAR	NO.	CODE
	32	0000000000000000
!	33	0010101010100010
"	34	0028282800000000
#	35	0028287C287C2828
$	36	0038545038145438
%	37	00606408102040C0C
&	38	0020505020544834
'	39	0008081000000000
(	40	0008102020201008
)	41	0020100808081020
*	42	000028107C102800
+	43	000010107C101000
,	44	0000000000301020
-	45	000000007C000000
.	46	0000000000003030
/	47	0000040810204000
0	48	0038444444444438
1	49	0010301010101038
2	50	003844040810207C
3	51	0038440418044438
4	52	00081828487C0808
5	53	007C407804044438
6	54	0018204078444438
7	55	007C040810202020
8	56	0038444438444438
9	57	003844443C040830
:	58	0000303000303000
;	59	0000303000301020
<	60	0008102040201008
=	61	0000007C007C0000
>	62	0020100804081020
?	63	0038440408100010
@	64	0038445C545C4038
A	65	003844447C444444
B	66	0078242438242478
C	67	0038444040404438
D	68	0078242424242478
E	69	007C40407840407C
F	70	007C404078404040
G	71	003C40405C444438
H	72	004444447C444444
I	73	0038101010101038
J	74	0004040404044438
K	75	0044485060504844
L	76	004040404040407C
M	77	00446C5454444444
N	78	00446464544C4C44
O	79	007C44444444447C
P	80	0078444478404040
Q	81	0038444444544834
R	82	0078444478504844
S	83	0038444038044438
T	84	007C101010101010
U	85	0044444444444438
V	86	0044444428281010
W	87	0044444454545428
X	88	0044442810284444
Y	89	0044442810101010
Z	90	007C04081020407C
[	91	0038202020202038
\	92	0000402010080400
]	93	0038080808080838
^	94	0000102844000000
	95	000000000000007C
`	96	0000201008000000
a	97	0000003844 7C4444
b	98	0000007824382478
c	99	0000003C4040403C
d	100	0000007824242478
e	101	0000007C4078407C
f	102	0000007C40784040
g	103	0000003C405C4438
h	104	00000044447C4444
i	105	0000003810101038
j	106	0000000808084830
k	107	0000002428302824
l	108	000000404040407C
m	109	000000446C544444
n	110	0000004464544C44
o	111	0000007C4444447C
p	112	0000007844784040
q	113	0000003844544834
r	114	0000007844784844
s	115	0000003C40380478
t	116	0000007C10101010
u	117	0000004444444438
v	118	0000004444282810
w	119	0000004444545428
x	120	0000004428102844
y	121	0000004428101010
z	122	0000007C0810207C
{	123	0018202040202018
|	124	0010101000101010
}	125	0030080804080830
~	126	0000205408000000
	127	0000000000000000

```
*********************
*  PATIENCE PLEASE  *
*     V-PI531KB     *
*    BY T CASTLE    *
*********************
```

DESCRIPTION. We're not sure if this particular version of Solitaire has a name or not, but it's more challenging, and affords you a greater opportunity to win, than the traditional game known as "Klondike". The game utilizes a standard 52 card playing deck. Twenty eight cards are initially dealt into seven piles. Dealing is crosswise, with one card being dealt to each pile and one less pile being dealt to each time. The first card in each round of dealing is turned up. After seven rounds, the first pile on the left will have one card turned face up and the pile on the far right will have six cards down and one card face up. After this initial deal, the remaining 24 cards are all dealt face up, in four rounds, dealing crosswise from the second through seventh pile. You won't actually have to worry about this, since this program completes the deal automatically and displays all of the up cards in all seven piles. Each card is numbered in either red or black and is displayed with the appropriate suit (Diamond, Heart, Club, or Spade symbol), using the codes developed in Chapter 4. The entire tableau is displayed on a dark green background. We don't have the ability to print the symbols here, so we'll use a H, D, C, and S to represent the Heart, Diamond, Club and Spade. A typical tableau might look like this.

A-H	5-D	Q-S	K-C	A-C	1Ø-C	Q-C
	4-D	9-S	K-D	A-D	8-D	5-C
	1Ø-S	4-H	2-H	J-S	Q-D	1Ø-D
	3-H	6-H	2-S	J-D	8-C	2-C
	7-C	9-H	6-C	8-S	K-H	7-H

Rules. After the deal, as you can see, the cards are not in any kind of order in the individual piles. From here on, movement is in the traditional "Solitaire" style. You must move red to black or black to red and the piles must be built in descending order with King (high) and Ace (low). A King may not be built on an Ace. The card that you are moving to must be an exposed card (the uppermost in any pile). In the example, the A-H, 7-C, 9-H, 6-C, 8-S, K-H, and 7-H are eligible. You may move any up card to one of these cards, provided it's black on red or red on black, and it's in descending order. All cards below that card are also moved. In the example, the 5-D (first card, second pile) can be moved to the 6-C (last card, fourth pile). The entire stack is moved by the computer and, after it's moved, the down card under the 5-D is turned over. Continue to move cards around, attempting to get them in order, and expose all down cards. As in traditional solitaire, Ace's are played up and King's are moved to open piles (where all down cards have been turned over and used). The up cards (built above the tableau) are built in ascending order with all cards of one suit in each pile. In our example, the A-H can be played up since it has no cards beneath it. When the 2-H is exposed it can be moved up.

Controlling Movement. You control the movement of the cards with the keyboard. To make the first move mentioned above, key in 5D and 6C. You don't have to hit the enter key. Your move will be indicated in the upper left portion of the screen as you key it in as follows: "5D-6C". After a slight pause, while the computer checks the validity of your move, the cards will move as indicated. All cards are represented as two digit codes, e.g. two of

Hearts is 2H, Ace of Diamonds is 1D, Jack of Spades is JS, and ten of Clubs is ØC. To move a King to a blank space enter the code for the King, followed by the letter "M", e.g. KH-M. To move a card to the upper piles, enter the code for the card, followed by the letter "U", e.g. AH-U. Don't worry if you enter it wrong, since this program won't let you cheat or make a mistake. It checks sequence, black & white, whether or not there is a blank space to move to, or whether a card is eligible for movement. There are three other options listed in the upper left corner of the screen. FCTN-3 erases the current entry if you realize you made a mistake before completing it. FCTN-5 deals a new game. FCTN-8 redeals the deck in exactly the same order. There is a specific reason for this option. Because of the random order in which these cards are placed, often times you'll have two or three choices of moves. Depending on what order you move them in, you may win or lose the game. By redealing the cards exactly the same, you can try a different strategy. We've been told that this game can be won every time.

NOTES. The basic sequence of operation in this program is contained in lines 17Ø through 32Ø. The general character coding, creation of the deck and the shuffle are found in subroutines beginning at 39Ø and 96Ø. What you do with them, after they are created, will vary from game to game, depending on your needs. In this particular game, in addition to the DECK$ which is originally created, we utilize three other arrays to keep track of where all the cards are. TBU$(X,Y) is a two dimensional array where; X represents the pile, and Y represents the up cards relative position in that file. TBD$(X,Y) represents the down cards where; X is the pile, and Y is the relative position in the pile. ELG$ is a one dimensional array which keeps track of all cards which are either eligible for movement to one of the upper piles or which are eligible to have another card moved to them. The values represented in these arrays are actually 6 digit codes, where the first three digits represent the ASC character value of the number and the second group of three represent the ASC character value of the Suit. In other words, a typical value for TBU$(2,2) might be "131143". "131" is the redefined character code for a red four. "143" is the redefined character code for a red diamond. By storing this information as a character code, it facilitates easy printing to any position specified.

After you enter your move, using the CALL KEY subroutine, the first thing the computer does is convert your move to ASC codes. Depending on whether you enter two card values, a card value followed by an "M", or a card value followed by a "U", the computer uses the codes to determine if it's a legal move. If a move isn't legal, one of the error messages, shown in 75Ø through 82Ø, is selected and printed to the screen. If it's legal, the computer moves the cards and changes the appropriate values in TBU$ (up cards), TBD$ (down cards), and ELG$ (eligible cards). The subroutine in lines 112Ø to 122Ø is used to print all cards to the screen.

```
100 REM  *******************
110 REM  * PATIENCE PLEASE *
120 REM  *******************
130 REM
140 REM  BY T CASTLE
150 REM  AMLIST V-PI531KB
160 REM
170 REM  GENERAL START DATA
180 GOSUB 340
190 REM  SPECIAL GAME DATA
200 GOSUB 750
210 REM  SHUFFLE CARDS
220 GOSUB 960
230 REM  DEAL CARDS
240 GOSUB 1240
250 GOSUB 4340
260 REM  CALL KEY INPUT
270 GOSUB 1800
280 IF KY=6 THEN 240
290 IF KY=14 THEN 220
300 REM  VERIFY & MOVE
310 GOSUB 2070
320 GOTO 270
330 REM  STARTING DATA
340 CALL CLEAR
350 CALL SCREEN(13)
360 DIM DECK$(52),TBU$(7,15)
370 DIM TBD$(7,7),ELG$(52)
380 DIM CODE$(83)
390 REM  A,2,3,4,5,6,7,8,9,J
,Q,K
400 DATA FF3844447C444444,FF
3844040810207C
410 DATA FF38440418044438,FF
081828487C0808
420 DATA FF7C407804044438,FF
1820407844438
430 DATA FF7C040810202020,FF
3844443844438
440 DATA FF3844443C040830,FF
4C52525252524C
450 DATA FF04040404044438,FF
38444444544834
460 DATA FF44485060504844
470 FOR J=128 TO 144 STEP 16
480 RESTORE 400
490 FOR I=J TO J+12
500 READ A$
510 CALL CHAR(I,A$)
520 NEXT I
530 NEXT J
540 CALL CHAR(142,"FFC6EEFEF
E7C3810")
550 CALL CHAR(143,"FF10387CF
E7C3810")
560 CALL CHAR(158,"FF3838FEF
EFE107C")
570 CALL CHAR(159,"FF10387CF
EFED638")
580 CALL CHAR(141,"FF0")
590 CALL CHAR(157,"0")
600 FOR K=128 TO 144 STEP 16
610 FOR J=K+14 TO K+15
620 FOR I=K TO K+12
630 NUMB$=STR$(I)&STR$(J)
640 L=L+1
650 DECK$(L)=NUMB$
660 NEXT I
670 NEXT J
680 NEXT K
690 CALL COLOR(13,7,16)
700 CALL COLOR(14,7,16)
710 CALL COLOR(15,2,16)
720 CALL COLOR(16,2,16)
730 RETURN
740 REM  SPECIAL GAME CODES
750 MSG$(1)="NOT FOUND "
760 MSG$(2)="INELIGIBLE"
770 MSG$(3)="NO SPACE  "
780 MSG$(4)="BAD VALUE "
790 MSG$(5)="RED/BLACK "
800 MSG$(6)="SEQUENCE  "
810 MSG$(7)="TOO LONG  "
820 MSG$(8)="SAME ROW  "
830 CODE$(68)="143"
840 CODE$(72)="142"
850 CODE$(67)="158"
860 CODE$(83)="159"
870 FOR I=49 TO 57
880 CODE$(I)=STR$(I+79)
890 NEXT I
900 CODE$(48)="137"
910 CODE$(74)="138"
920 CODE$(81)="139"
930 CODE$(75)="140"
940 RETURN
950 REM    THE SHUFFLE
960 CALL CLEAR
```

```
970 Z=Z+1
980 RANDOMIZE
990 FOR I=1 TO 52
1000 F1=INT(52*RND)+1
1010 F1$=DECK$(F1)
1020 F2=INT(52*RND)+1
1030 IF F2=F1 THEN 1020
1040 F2$=DECK$(F2)
1050 DECK$(F2)=F1$
1060 DECK$(F1)=F2$
1070 NEXT I
1080 IF Z<2 THEN 960
1090 Z=2
1100 RETURN
1110 REM  PRINT CARDS
1120 CALL SOUND(150,1450,0)
1130 SUIT=VAL(SEG$(CARD$,4,3
))
1140 CRD=VAL(SEG$(CARD$,1,3)
)
1150 CALL HCHAR(ROW,COL,CRD)
1160 CALL HCHAR(ROW,COL+1,SU
IT)
1170 CALL HCHAR(ROW,COL+2,14
1)
1180 CALL HCHAR(ROW+1,COL,15
7,3)
1190 CALL HCHAR(ROW+2,COL,15
7,3)
1200 CALL HCHAR(ROW+3,COL,15
7,3)
1210 CALL HCHAR(ROW+4,COL,15
7,3)
1220 RETURN
1230 REM  THE DEAL
1240 CALL CLEAR
1250 RESTORE 1260
1260 DATA 127,127,143,143,0
1270 READ ACE$(1),ACE$(2),AC
E$(3),ACE$(4),ADE
1280 FOR I=1 TO 52
1290 ELG$(I)=""
1300 IF I>7 THEN 1360
1310 FOR K=1 TO 15
1320 IF K>7 THEN 1340
1330 TBD$(I,K)=""
1340 TBU$(I,K)=""
1350 NEXT K
1360 NEXT I
1370 DATA 1,47,48,49,50,51,5
2
1380 FOR I=1 TO 7
1390 READ A
1400 IF I>1 THEN 1430
1410 TBU$(I,1)=DECK$(A)
1420 GOTO 1440
1430 TBU$(I,5)=DECK$(A)
1440 ADE=ADE+1
1450 ELG$(ADE)=DECK$(A)
1460 NEXT I
1470 DATA 2,3,9,4,10,15,5,11
,16,20,6,12,17,21,24,7,13,18
,22,25,27
1480 FOR I=2 TO 7
1490 FOR K=1 TO I-1
1500 READ A
1510 TBD$(I,K)=DECK$(A)
1520 NEXT K
1530 NEXT I
1540 DATA 8,14,19,23,26,28
1550 FOR I=2 TO 7
1560 READ A
1570 TBU$(I,1)=DECK$(A)
1580 NEXT I
1590 J=1
1600 FOR K=29 TO 41 STEP 6
1610 J=J+1
1620 FOR I=2 TO 7
1630 TBU$(I,J)=DECK$(K+I-2)
1640 NEXT I
1650 NEXT K
1660 ROW=7
1670 COL=3
1680 CARD$=TBU$(1,1)
1690 GOSUB 1120
1700 FOR K=1 TO 5
1710 ROW=6+K
1720 FOR I=2 TO 7
1730 COL=(I*4)-1
1740 CARD$=TBU$(I,K)
1750 GOSUB 1120
1760 NEXT I
1770 NEXT K
1780 RETURN
1790 REM  CALL KEY
1800 CALL HCHAR(1,1,32,11)
1810 RESTORE 1820
1820 DATA 0,0,,,,,
```

```
1830 READ J,J1,MV1$,MV2$,MV$
,CKX$
1840 CALL KEY(3,KY,ST)
1850 IF ST<1 THEN 1840
1860 IF (KY=6)+(KY=14)THEN 2
050
1870 IF KY=7 THEN 1800
1880 J=J+1
1890 CALL HCHAR(1,1+J,KY)
1900 MV$=MV$&STR$(KY)
1910 IF J1=0 THEN 1940
1920 IF MV$="77" THEN 1950
1930 IF MV$="85" THEN 1950
1940 IF LEN(MV$)<4 THEN 1840
1950 J1=J1+1
1960 IF J1>1 THEN 2000
1970 MV1$=MV$
1980 MV$=""
1990 GOTO 2020
2000 MV2$=MV$
2010 GOTO 2050
2020 CALL HCHAR(1,4,45)
2030 J=3
2040 GOTO 1840
2050 RETURN
2060 REM  VALIDATES DATA
2070 REM  DO MV1$
2080 MV$=MV1$
2090 GOSUB 2720
2100 IF CKX$="X" THEN 2700
2110 CK1$=CK$
2120 IF MV2$="77" THEN 2400
2130 IF MV2$="85" THEN 2540
2140 MV$=MV2$
2150 GOSUB 2720
2160 IF CKX$="X" THEN 2700
2170 CK2$=CK$
2180 GOSUB 2920
2190 IF CKX$="X" THEN 2700
2200 CK$=CK2$
2210 GOSUB 3070
2220 IF CKX$="X" THEN 2700
2230 CK$=CK1$
2240 GOSUB 3180
2250 IF CKX$="X" THEN 2700
2260 PR1=PR
2270 PS1=PS
2280 PE1=PE
2290 CK$=CK2$
2300 GOSUB 3180
2310 IF CKX$="X" THEN 2700
2320 PR2=PR
2330 PS2=PS
2340 PE2=PE
2350 GOSUB 3390
2360 IF CKX$="X" THEN 2700
2370 GOSUB 3470
2380 GOSUB 3710
2390 GOTO 2700
2400 CK$=CK1$
2410 GOSUB 3180
2420 IF CKX$="X" THEN 2700
2430 PR1=PR
2440 PS1=PS
2450 PE1=PE
2460 PS2=0
2470 GOSUB 3890
2480 IF CKX$="X" THEN 2700
2490 PR2=SP
2500 GOSUB 3470
2510 GOSUB 3720
2520 GOTO 2700
2530 REM  MOVE ACES
2540 ACECK=1
2550 CK$=CK1$
2560 GOSUB 3070
2570 IF CKX$="X" THEN 2700
2580 GOSUB 3180
2590 IF CKX$="X" THEN 2700
2600 PR1=PR
2610 PS1=PS
2620 PE1=PE
2630 GOSUB 4060
2640 IF CKX$="X" THEN 2700
2650 PS2=-6
2660 GOSUB 3470
2670 GOSUB 3710
2680 ACE$(PR2-3)=CK1$
2690 ACECK=0
2700 RETURN
2710 REM  CONVERT STRING
2720 IF LEN(MV$)<>4 THEN 280
0
2730 C1=VAL(SEG$(MV$,1,2))
2740 IF (C1=74)+(C1=75)THEN
2770
2750 IF C1=81 THEN 2770
```

```
2760 IF (C1<48)+(C1>57)THEN
2800
2770 C2=VAL(SEG$(MV$,3,2))
2780 IF (C2=68)+(C2=67)THEN
2830
2790 IF (C2=72)+(C2=83)THEN
2830
2800 TYP=4
2810 GOSUB 4280
2820 GOTO 2900
2830 CK$=CODE$(C2)
2840 IF VAL(CK$)>143 THEN 28
70
2850 CK$=CODE$(C1)&CK$
2860 GOTO 2900
2870 TCK=VAL(CODE$(C1))+16
2880 TCK$=STR$(TCK)
2890 CK$=TCK$&CK$
2900 RETURN
2910 REM   CHK RD/BLK&SEQ
2920 K1=VAL(SEG$(CK1$,4,3))
2930 K2=VAL(SEG$(CK2$,4,3))
2940 K3=K1-K2
2950 IF (K3<-1)+(K3>1)THEN 2
980
2960 TYP=5
2970 GOTO 3040
2980 K2=VAL(SEG$(CK2$,1,3))
2990 K1=VAL(SEG$(CK1$,1,3))
3000 K3=K2-K1
3010 IF K3=-15 THEN 3050
3020 IF K3=17 THEN 3050
3030 TYP=6
3040 GOSUB 4280
3050 RETURN
3060 REM  CHECK ELIG
3070 SEL=0
3080 FOR I=1 TO 12
3090 IF CK$<>ELG$(I)THEN 312
0
3100 SEL=I
3110 I=12
3120 NEXT I
3130 IF SEL>0 THEN 3160
3140 TYP=2
3150 GOSUB 4280
3160 RETURN
3170 REM  CHECKS TO FIND
3180 DATA 0,0,0,0
3190 RESTORE 3180
3200 READ PR,PS,PE,CKX$
3210 FOR I=1 TO 7
3220 FOR K=1 TO 15
3230 IF PR=0 THEN 3290
3240 IF TBU$(I,K)<>"" THEN 3
320
3250 PE=K-1
3260 K=15
3270 I=7
3280 GOTO 3320
3290 IF CK$<>TBU$(I,K)THEN 3
320
3300 PR=I
3310 PS=K
3320 NEXT K
3330 NEXT I
3340 IF PR>0 THEN 3370
3350 TYP=1
3360 GOSUB 4280
3370 RETURN
3380 REM   CHK ROW & LENGTH
3390 IF PR1<>PR2 THEN 3420
3400 TYP=8
3410 GOSUB 4280
3420 IF PS2+(PE1-PS1)+1<15 T
HEN 3450
3430 TYP=7
3440 GOSUB 4280
3450 RETURN
3460 REM  ERASE & MOVE
3470 COL=(PR1*4)-1
3480 IF PS1>1 THEN 3530
3490 FOR I=PS1+6 TO 24
3500 CALL HCHAR(I,COL,32,3)
3510 NEXT I
3520 GOTO 3590
3530 FOR I=PS1+6 TO PS1+9
3540 CALL HCHAR(I,COL,157,3)
3550 NEXT I
3560 FOR I=PS1+10 TO 24
3570 CALL HCHAR(I,COL,32,3)
3580 NEXT I
3590 COL=(PR2*4)-1
3600 FOR I=PS1 TO PE1
3610 CARD$=TBU$(PR1,I)
3620 ROW=(I-PS1)+(7+PS2)
3630 GOSUB 1120
```

```
3640 TBU$(PR1,I)=""
3650 IF ACECK=1 THEN 3680
3660 N=(I-PS1)+PS2+1
3670 TBU$(PR2,N)=CARD$
3680 NEXT I
3690 RETURN
3700 REM  CHANGE ELIG
3710 ELG$(SEL)=""
3720 IF PS1<2 THEN 3750
3730 ELG$(SEL)=TBU$(PR1,PS1-
1)
3740 GOTO 3870
3750 IF TBD$(PR1,1)="" THEN
3870
3760 FOR I=1 TO 7
3770 IF TBD$(PR1,I)<>"" THEN
 3860
3780 CARD$=TBD$(PR1,I-1)
3790 TBU$(PR1,1)=CARD$
3800 ELG$(SEL)=CARD$
3810 TBD$(PR1,I-1)=""
3820 COL=(PR1*4)-1
3830 ROW=7
3840 GOSUB 1120
3850 I=7
3860 NEXT I
3870 RETURN
3880 REM   FIND SPOT FOR K
3890 SP=0
3900 FOR I=1 TO 7
3910 IF TBU$(I,1)<>"" THEN 3
940
3920 SP=I
3930 I=7
3940 NEXT I
3950 IF SP>0 THEN 3990
3960 TYP=3
3970 GOSUB 4280
3980 GOTO 4040
3990 FOR I=1 TO 7
4000 IF ELG$(I)<>"" THEN 403
0
4010 SEL=I
4020 I=7
4030 NEXT I
4040 RETURN
4050 REM  LOCATE UP CARDS
4060 C1=VAL(SEG$(CK1$,1,3))
4070 C2=VAL(SEG$(CK1$,4,3))
4080 IF C2=142 THEN 4120
4090 IF C2=143 THEN 4150
4100 IF C2=158 THEN 4180
4110 IF C2=159 THEN 4210
4120 C3=VAL(SEG$(ACE$(1),1,3
))
4130 PR2=4
4140 GOTO 4230
4150 C3=VAL(SEG$(ACE$(2),1,3
))
4160 PR2=5
4170 GOTO 4230
4180 C3=VAL(SEG$(ACE$(3),1,3
))
4190 PR2=6
4200 GOTO 4230
4210 C3=VAL(SEG$(ACE$(4),1,3
))
4220 PR2=7
4230 IF C1-C3=1 THEN 4260
4240 TYP=6
4250 GOSUB 4280
4260 RETURN
4270 REM  ERROR MESSAGE
4280 CKX$="X"
4290 FOR I=1 TO 10
4300 J=ASC(SEG$(MSG$(TYP),I,
1))
4310 CALL HCHAR(1,1+I,J)
4320 NEXT I
4330 RETURN
4340 MS$(1)="FCTN-8=SAME"
4350 MS$(2)="FCTN-5=NEW"
4360 MS$(3)="FCTN-3=ERASE"
4370 FOR I=1 TO 3
4380 L=LEN(MS$(I))
4390 FOR J=1 TO L
4400 LTR=ASC(SEG$(MS$(I),J,1
))
4410 CALL HCHAR(2+I,J+1,LTR)
4420 NEXT J
4430 NEXT I
4440 RETURN
```

HAPPY COMPUTING!

```
*****************
*  SUPER MAZE   *
*   V-PG431KB   *
*  BY T CASTLE  *
*****************
```

DESCRIPTION. Because it's different every time (unless you tell it to replay) this simple maze game will entertain the children for hours. After entering RUN it'll take about 2 1/2 minutes for the computer to construct a random maze and display it on the screen. The maze itself is white on a green background and the entire 22X3Ø grid has a border in red. A small plus sign (+) appears in the top row and represents the starting point, while a white "F" on a red background is found in the bottom row, representing the "Finish". Across the top (white on red letters) it reads "SCORE: TI- KEYS- ANS-". After the maze is displayed, a number generally between 85 and 12Ø, is displayed next to the TI. This represents the number of moves required to complete the maze if the computer solution is followed. Since movement through the maze is through use of the up, down, left, and right arrow keys, the computer will count and display the number of key strokes you use as the game progresses. If you run into a dead end and need to backtrack, the computer will erase your track as you go backwards while the number of "KEYS" keeps increasing. When you reach the "F" the key counter stops counting, you get a ting-a-ling for your effort, and the number of moves required for your final solution is shown next to "ANS-" on the top line. Across the the bottom the user is given three choices: R for replay, A for answer, or N for new. At this point, or at any other point during the game, hitting an "A" will display the computer solution as yellow blocks with white dots. If you took exactly the same path, no yellow blocks would be visible; otherwise, any difference between your solution and the computer solution will be shown. If you want to play the same maze again to see if you can beat your score, or to compete against someone else who hasn't seen your solution or the computer solution, simply enter "R" for replay. The board clears immediately but takes about two minutes to rebuild before being displayed again. The last option, "N", permits you to build a brand new maze.

NOTES. The general program sequence is found in lines 1ØØ-35Ø and setting of all initial variables, CALL COLOR's, CALL CHAR's, etc., in lines 37Ø-61Ø. The maze itself is actually stored as a two dimensional array called MZ1(I,K) where "I" represents row and "K" represents column. The values stored in the array are the ASC character codes which are eventually printed to the screen. Character codes 144 and 145 represent the solution, while other codes represent spaces, lines or other maze configurations. Building the maze is a three part process which begins with constructing a solution (lines 132Ø-168Ø). Next, in lines 63Ø-75Ø, the computer randomly places spaces and characters with the line to the left and bottom (carefully avoiding solution blocks). Lastly, in lines 76Ø-94Ø the computer fills in any other spaces, not already assigned, with a line. All of this information is placed and stored directly in the array MZ1. On the last pass, the computer also prints the character to the screen. In lines 95Ø-116Ø the border and messages are printed and the necessary CALL COLOR statements are performed to "turn the lights on" on the screen. The program then goes to 169Ø which controls the movement through the maze.

```
100 REM  **************
110 REM  * SUPER MAZE *
120 REM  **************
130 REM
140 REM  BY T CASTLE
150 REM  AMLIST V-PG431KB
160 REM
170 GOSUB 370
180 GOSUB 1330
190 GOSUB 630
200 GOSUB 1700
210 MSG$="0131"&STR$(ANSW)
220 GOSUB 2780
230 GOTO 260
240 CALL KEY(3,KY,ST)
250 IF ST=0 THEN 240
260 IF KY=65 THEN 300
270 IF KY=78 THEN 170
280 IF KY=82 THEN 340
290 GOTO 240
300 CALL CHAR(144,"00003C3C0
000FFFF")
310 CALL CHAR(145,"00003C3C"
)
320 CALL COLOR(15,16,11)
330 GOTO 240
340 GOSUB 1180
350 GOTO 200
360 REM  SET INITIAL VAR
370 CALL CLEAR
380 DIM MZ1(22,30)
390 FOR I=1 TO 15
400 CALL COLOR(I,4,4)
410 NEXT I
420 DATA 128,808080808080FFF
F,129,000000000000FFFF
430 DATA 130,808080808080808
0,131,0
440 DATA 132,0010107C1010FFF
F,133,809090FC9090FFFF
450 DATA 134,0010107C101,135
,809090FC90908080
460 DATA 136,FFFFFFFFFFFFFFF
F,144,000000000000FFFF
470 DATA 145,0
480 RESTORE 420
490 FOR I=1 TO 11
500 READ A,B$
510 CALL CHAR(A,B$)
520 NEXT I
530 CSOL=-1
540 ANSW=1
550 KMOV=0
560 FOR I=1 TO 22
570 FOR J=1 TO 30
580 MZ1(I,J)=0
590 NEXT J
600 NEXT I
610 RETURN
620 REM   CREATES MAZE
630 RANDOMIZE
640 FOR J=1 TO 22
650 FOR I=1 TO 29 STEP 7
660 FOR K=130 TO 131
670 N=INT(((I+7)-I+1)*RND)+I
680 IF N>30 THEN 730
690 X=MZ1(J,N)
700 IF X=130 THEN 670
710 IF (X=144)+(X=145)THEN 7
30
720 MZ1(J,N)=K
730 NEXT K
740 NEXT I
750 NEXT J
760 FOR I=1 TO 22
770 FOR J=1 TO 30
780 X=MZ1(I,J)
790 IF (X=144)+(X=145)THEN 9
10
800 IF X=0 THEN 890
810 IF (X=128)+(X=130)THEN 9
20
820 IF I<13 THEN 920
830 IF (X=131)+(X1=1)THEN 88
0
840 X1=1
850 IF J-1<1 THEN 870
860 IF MZ1(I,J-1)=145 THEN 8
90
870 GOTO 920
880 X1=0
890 MZ1(I,J)=129
900 GOTO 920
910 CSOL=CSOL+1
920 CALL HCHAR(I+1,J+1,MZ1(I
,J))
```

```
930 NEXT J
940 NEXT I
950 CALL VCHAR(1,1,136,24)
960 CALL VCHAR(1,32,136,24)
970 CALL HCHAR(1,1,136,32)
980 CALL HCHAR(24,1,136,32)
990 MSG$="0103SCORE: TI-"&ST
R$(CSOL)
1000 MSG$=MSG$&" KEYS-     A
NS-"
1010 GOSUB 2780
1020 MSG$="2403R-REPLAY    A
-ANSWER     N-NEW"
1030 GOSUB 2780
1040 FOR I=1 TO 12
1050 CALL COLOR(I,16,7)
1060 NEXT I
1070 CALL COLOR(14,7,7)
1080 CALL COLOR(13,16,4)
1090 CALL COLOR(15,16,4)
1100 CALL GCHAR(SR,SC,NR)
1110 IF NR=144 THEN 1140
1120 CALL HCHAR(SR,SC,134)
1130 GOTO 1150
1140 CALL HCHAR(SR,SC,132)
1150 CALL HCHAR(FR,FC,70)
1160 RETURN
1170 REM  REPLAY
1180 FOR I=1 TO 15
1190 CALL COLOR(I,4,4)
1200 NEXT I
1210 CALL CHAR(144,"00000000
0000FFFF")
1220 CALL CHAR(145,"0")
1230 FOR I=1 TO 22
1240 FOR J=1 TO 30
1250 CALL HCHAR(I+1,J+1,MZ1(
I,J))
1260 NEXT J
1270 NEXT I
1280 ANSW=1
1290 KMOV=0
1300 GOSUB 950
1310 RETURN
1320 REM  CREATES SOLOUTION
1330 CALL CLEAR
1340 RANDOMIZE
1350 J=1
1360 K=INT(11*RND)+10
1370 SR=2
1380 SC=K+1
1390 L=INT(2*RND)+1
1400 IF J+1=23 THEN 1660
1410 IF J+1+L>22 THEN 1420 E
LSE 1430
1420 L=22-J
1430 FOR I=J TO J+L-1
1440 MZ1(I,K)=145
1450 NEXT I
1460 MZ1(I,K)=144
1470 J=J+L
1480 L=INT(10*RND)+1
1490 D=INT(2*RND)+1
1500 IF D=1 THEN 1580
1510 IF K+L>30 THEN 1580
1520 FOR G=K TO K+L-1
1530 MZ1(J,G)=144
1540 NEXT G
1550 MZ1(J,G)=145
1560 K=K+L
1570 GOTO 1390
1580 IF K-L<2 THEN 1510
1590 FOR G=K TO K-L+1 STEP -
1
1600 MZ1(J,G)=144
1610 NEXT G
1620 MZ1(J,G)=145
1630 K=K-L
1640 GOTO 1390
1650 MZ1(22,K+1)=144
1660 FR=23
1670 FC=K+1
1680 RETURN
1690 REM  CALL KEY MOVEMENT
1700 R=SR
1710 C=SC
1720 CALL KEY(3,KY,ST)
1730 IF ST=0 THEN 1720
1740 KMOV=KMOV+1
1750 MSG$=STR$(KMOV)
1760 IF KMOV<10 THEN 1800
1770 IF KMOV<100 THEN 1790
1780 CALL HCHAR(1,23,ASC(SEG
$(MSG$,3,1)))
1790 CALL HCHAR(1,22,ASC(SEG
$(MSG$,2,1)))
```

```
1800 CALL HCHAR(1,21,ASC(SEG
$(MSG$,1,1)))
1810 CALL SOUND(10,500,0)
1820 REM  CHECK UP
1830 IF KY<>69 THEN 2020
1840 CALL GCHAR(R-1,C,NR)
1850 IF NR=70 THEN 2720
1860 IF (NR=130)+(NR=131)THE
N 1890
1870 IF NR=145 THEN 1890
1880 IF (NR=134)+(NR=135)THE
N 1920 ELSE 1720
1890 R=R-1
1900 IF NR=130 THEN 1960
1910 IF (NR=131)+(NR=145)THE
N 1980
1920 CALL HCHAR(R,C,MZ1(R-1,
C-1))
1930 R=R-1
1940 ANSW=ANSW-1
1950 GOTO 1720
1960 CALL HCHAR(R,C,135)
1970 GOTO 1990
1980 CALL HCHAR(R,C,134)
1990 ANSW=ANSW+1
2000 GOTO 1720
2010 REM  DOWN
2020 IF KY<>88 THEN 2270
2030 CALL GCHAR(R,C,NR)
2040 IF (NR=132)+(NR=133)THE
N 1720
2050 CALL GCHAR(R+1,C,NR1)
2060 IF (NR1=136)+(NR1<128)T
HEN 1720
2070 R=R+1
2080 CALL GCHAR(R,C,NR)
2090 IF NR=70 THEN 2720
2100 IF (NR=144)+(NR=129)THE
N 2170
2110 IF (NR=145)+(NR=131)THE
N 2210
2120 IF NR=128 THEN 2190
2130 IF NR=130 THEN 2230
2140 CALL HCHAR(R-1,C,MZ1(R-
2,C-1))
2150 ANSW=ANSW-1
2160 GOTO 1720
2170 CALL HCHAR(R,C,132)
2180 GOTO 2240
2190 CALL HCHAR(R,C,133)
2200 GOTO 2240
2210 CALL HCHAR(R,C,134)
2220 GOTO 2240
2230 CALL HCHAR(R,C,135)
2240 ANSW=ANSW+1
2250 GOTO 1720
2260 REM  RIGHT
2270 IF KY<>68 THEN 2450
2280 CALL GCHAR(R,C+1,NR)
2290 IF (NR=128)+(NR=130)THE
N 1720
2300 IF (NR=133)+(NR=135)THE
N 1720
2310 IF NR=136 THEN 1720
2320 C=C+1
2330 IF NR=70 THEN 2720
2340 IF (NR=144)+(NR=129)THE
N 2390
2350 IF (NR=145)+(NR=131)THE
N 2410
2360 CALL HCHAR(R,C-1,MZ1(R-
1,C-2))
2370 ANSW=ANSW-1
2380 GOTO 1720
2390 CALL HCHAR(R,C,132)
2400 GOTO 2420
2410 CALL HCHAR(R,C,134)
2420 ANSW=ANSW+1
2430 GOTO 1720
2440 REM  LEFT
2450 IF KY<>83 THEN 2700
2460 CALL GCHAR(R,C,NR)
2470 IF (NR=133)+(NR=135)THE
N 1720
2480 IF (NR=128)+(NR=130)THE
N 1720
2490 CALL GCHAR(R,C-1,NR1)
2500 IF NR1=136 THEN 1720
2510 C=C-1
2520 CALL GCHAR(R,C,NR)
2530 IF NR=70 THEN 2720
2540 IF (NR=144)+(NR=129)THE
N 2610
2550 IF (NR=145)+(NR=131)THE
N 2650
2560 IF NR=128 THEN 2630
```

```
2570 IF NR=130 THEN 2670
2580 CALL HCHAR(R,C+1,MZ1(R-
1,C))
2590 ANSW=ANSW-1
2600 GOTO 1720
2610 CALL HCHAR(R,C,132)
2620 GOTO 2680
2630 CALL HCHAR(R,C,133)
2640 GOTO 2680
2650 CALL HCHAR(R,C,134)
2660 GOTO 2680
2670 CALL HCHAR(R,C,135)
2680 ANSW=ANSW+1
2690 GOTO 1720
2700 IF (KY=65)+(KY=78)THEN
2760
2710 IF KY=82 THEN 2760 ELSE
 1720
2720 FOR I=1 TO 10
2730 CALL SOUND(80,900,0)
2740 CALL SOUND(80,600,0)
2750 NEXT I
2760 RETURN
2770 REM  MSG PRINT
2780 FOR I=1 TO LEN(MSG$)-4
2790 LTR=ASC(SEG$(MSG$,I+4,1
))
2800 ROW=VAL(SEG$(MSG$,1,2))
2810 COL=VAL(SEG$(MSG$,3,2))
2820 CALL HCHAR(ROW,COL+I-2,
LTR)
2830 NEXT I
2840 RETURN
```

HAPPY COMPUTING!

CHAPTER FIVE

Sound Effects & Music

GENERAL. This is probably the hardest part of all to write about because you really have to HEAR it to appreciate it. That's why this chapter will mostly be a collection of samples that you can try yourself. Experimentation is really the key to getting good sound effects.

We do have a few pointers and tricks that may help you in your experimentation process. Following are a few sample routines just to get you started. In some cases we've provided a little graphics to make it more interesting.

Going Up (and Down). This produces a red band, moving up the center of the screen, and shows the value of using negative values to cover movement on the screen. Enter the following:

```
>100 FR=300
>110 CALL CHAR(128,"FFFFFFFFF
 FFFFFFF")
>120 CALL COLOR(13,7,1)
>130 CALL CLEAR
>140 FOR I=24 TO 1 STEP -1
>150 FR=FR+10
>160 CALL SOUND(-100,FR,0)
>170 CALL HCHAR(I,16,128)
>180 NEXT I
>190 GOTO 190
```

There are several things that you can learn from this example. First, RUN it as it is. You'll hear a rather steady tone, increasing in frequency from 300 to about 540, as it moves up the screen. Now, without changing anything else, simply change line 150 to FR=FR+50, and RUN it again. You obviously go to a higher frequency, but you also begin to hear perceptible changes in the tone which breaks the smoothness of the program. Lesson - Somewhere between 10 and 50 is the limit for frequency changes if you want gradual movement up a scale.

Now change the FR statement back to an increment of 10 and change the last variable in the CALL SOUND statement (the 0) to 1.25*I. Here we make use of the variable in the loop to increase the volume from 30 (1.25*24) to 1.25 (1.25*1). Lesson - If you want a little variety in your programs, look at the values of other variables at the time a CALL SOUND is being made and use them to vary your sound. A CALL HCHAR to the left side of the screen has a low column value, while it has a high column value to the right. The value would be from 1 to 32 and almost corresponds to the range of volume. The same would hold true of the row values and is essentially what we used in the previous example.

Last, change the duration value from -100 to 100 and RUN the program. Now this creates a very definite break in the pattern and also slows the program down. The reason is, with a positive value, each CALL SOUND must be completed before the next is CALLed. In this case, each will last about 100 milliseconds or .1 of a second. Even if you lower this value to a positive 5, while it is no longer slow, it

still has a definite STOP, prior to doing the next CALL SOUND. With a negative number, it terminates immediately when it hits the next CALL SOUND, regardless of how long it has been playing. Try using negative sounds from -10 through -100 in increments of about -20 each time. You'll notice that, even with negative numbers, below -100, there is still a break in the sound. After a CALL SOUND is started, the computer continues on to other statements, such as the CALL HCHAR and movement through the loop. If the duration specified is not sufficient to cover the time necessary before it gets back to a new CALL SOUND, a break in the pattern will occur. Lesson - To have steady movement, up or down a scale, use negative numbers. Negative numbers give you the ability to "mask" other activities while sound is being created, yet they don't slow down the program because they terminate at the very next CALL SOUND.

For a variation on the above theme, change line 140 to FOR I=1 TO 24. Now you get the same thing in reverse, with the sound gradually fading away.

A Fitting Climax. The following example flashes your screen from color to color while a threatening sound builds to a climax. Enter the following:

```
>100 CALL SCREEN(11)
>110 CALL CLEAR
>120 FOR I=30 TO 0 STEP -3
>130 CALL SCREEN(4)
>140 CALL SOUND(-200,-6,I)
>150 CALL SCREEN(11)
>160 CALL SOUND(-100,-5,I)
>170 NEXT I
>180 GOTO 180
```

This would make a great finish to any game program. Leave your display on the screen, use compatible screen colors, and finish by printing "GAME OVER" in the middle of the screen. A good example of what the "White Noise" will do. Try this same thing with a -6 and -7, extend the times, and try different increments for the STEP command.

Radar. Good for submarine, battleship, and plane games.

```
>100 CALL CLEAR
>110 CALL SCREEN(15)
>120 J=200
>130 FOR I=0 TO 30 STEP 8
>140 CALL SCREEN(1)
>150 CALL SOUND(-70,2800-J,I)
>160 CALL SOUND(-70,2700-J,I)
>170 CALL SOUND(-10,9999,30)
>180 NEXT I
>190 FOR PAUSE=1 TO 150
>200 P=P+1
>210 NEXT PAUSE
>220 GOTO 110
```

This produces a black background which momentarily turns to a grey screen as the radar sounds. If you had ships or planes printed on the screen, they would be visible for just a second and then disappear when the screen goes black. You can extend the duration between "bleeps" by increasing the value in line 190. The way this is written the frequencies start at 2800 and 2700 and decrease by 200 each time it goes through the loop. The volume also decreases from a value of 0 (loud) to 30 (quiet). You can increase the amount of time the screen stays "turned on" by putting another pause statement in after line 110. By having a CALL SOUND in 150 and 160,

with a slight difference in frequency, we create a slight waver in the sound. The purpose of line 17Ø is to stop the last note from playing too long. The frequency is set above normal hearing range and volume is set to the quietest setting.

Shot in the Dark. We can't really explain this one, the title is probably the best description we can give it.

```
>1ØØ CALL SOUND(1ØØ,-5,Ø)
>11Ø CALL SOUND(2Ø,-5,Ø)
>12Ø FOR I=1 TO 3Ø STEP 3
>13Ø B=2ØØØ
>14Ø J=J+(.Ø2*B)
>15Ø CALL SOUND(-3ØØ,B-J,I,-5,
 I)
>16Ø NEXT I
```

Interesting effects can be created with this by adjusting the value of B. Try it at 3ØØØ, 1ØØØ, and 2ØØ. Also try adjusting the value in line 13Ø from .Ø2 either up or down. We think you'll agree that .Ø2 is the best value.

Test Program. The following program is one that we use as a starting point when looking for sounds. It starts out with some fixed values for Duration, Sound, Noise, and Tone. When it is first run, it displays these values. The program then runs continuously through a CALL KEY statement. By holding down the period (.) you can hear the sound with the values shown on the screen. Each time you hit the "D" the tone decreases by 5Ø. Each time you hit the "U" the tone increases by 5Ø. Starting value for Noise is -1. Hitting the "N" key changes this to -2, then -3, etc. It's good for experimentation purposes and, if you don't want to test noise, you can easily remove that portion by taking the 4th, and 5th values out of the CALL SOUND statement.

```
>1ØØ DUR=-5ØØ
>11Ø TON=5ØØ
>12Ø VOL=Ø
>13Ø NOI=-1
>14Ø GOSUB 32Ø
>15Ø CALL KEY(3,KY,ST)
>16Ø IF ST=Ø THEN 15Ø
>17Ø IF KY=46 THEN 3ØØ
>18Ø IF KY<>68 THEN 22Ø
>19Ø TON=TON-5Ø
>2ØØ GOSUB 32Ø
>21Ø GOTO 15Ø
>22Ø IF KY<>85 THEN 26Ø
>23Ø TON=TON+5Ø
>24Ø GOSUB 32Ø
>25Ø GOTO 15Ø
>26Ø IF KY<>78 THEN 15Ø
>27Ø NOI=NOI-1
>28Ø GOSUB 32Ø
>29Ø GOTO 15Ø
>3ØØ CALL SOUND(DUR,TON,VOL,N
 OI,VOL)
>31Ø GOTO 15Ø
>32Ø CALL CLEAR
>33Ø PRINT "DURATION= ",DUR
>34Ø PRINT "VOLUME   = ",VOL
>35Ø PRINT "NOISE    = ",NOI
>36Ø PRINT "TONE     = ",TON
>37Ø RETURN
```

Playing Songs. Knowing that the computer has the ability to create sounds and having a chart showing four octaves of notes, many of you may have been tempted to try to code in a complete song. The "Happy Birthday" program at the end of this chapter has just a small sampling of a song, but demonstrates some of the principles

involved in doing just that. Before going any further, let us add that a complete song can really be quite a task and presents some very real problems. It's not our purpose in this chapter to give a complete explanation of how to read sheet music, so we're going to assume you already know how or that you can find another book to teach you. All that's necessary is that you be able to read the notes for the melody and distinguish between a quarter note, eighth note, half note, whole note, etc. You also must know the meaning of the little dots following these notes which raise their value by 50%. For test purposes, we've managed to code in all of the words and the melody to "Frosty the Snowman". The words are displayed with a partially animated Frosty. Following is a discussion of the procedure.

First we had an initial section which set up a number of variables. We used the octave containing middle "C" as our starting point and made A=220, AS=233, B=247, C=262, etc. We had to get down to G in the lower octave so we coded any notes in this octave as GL=196, FL=175, etc. If we had to get above middle C we used an H following the note, such as BH=494, ASH=466. We also set up NN=9999 for a silent note. All that's necessary is that you create variables for all notes in the song you are coding in. Next we created a variable for each "duration" that we needed; however, we based it all on one figure — the length of one bar of music. We set up BASE=1000. This made our basic one bar of music approximately 1 second long. We then set up a different variable for each of the various types of notes, e.g, QN=.25*BASE,EN=.125*BASE, HN=.50*BASE, QNP=.375*BASE (meaning a 1/4 note raised an eighth). For loudness, we simply used a "0".

We then went through the sheet music and marked down these two variables (note and duration) by each of the notes shown for melody. When a "Rest" was indicated, we just used the proper duration code and the NN for tone. When we had this complete, we looked for a series of notes that repeated themselves. Many songs go through the same series of notes many times, with only the last one or two notes in that portion changing. We created subroutines for each of these repeating passages, and separate subroutines for each unusual ending or tag. The following is not a real song, but shows how a subroutine might look. It would represent about two complete bars of music.

```
>270 CALL SOUND(EN,B,0)
>280 CALL SOUND(EN,B,0)
>290 CALL SOUND(QN,C,0)
>300 CALL SOUND(QN,F,0)
>310 CALL SOUND(EN,C,0)
>320 CALL SOUND(QN,D,0)
>330 CALL SOUND(EN,E,0)
>340 CALL SOUND(HN,NN,0)
>350 RETURN
```

When all of this is done, you can write a main controlling section which runs the program through the appropriate passages and, between GOSUBS, picks up any additional notes that are missing or are unique and don't warrant a separate subroutine.

You may wonder why we don't code the numbers directly into the CALL SOUND statement, and why we don't use a DATA statement and a FOR-NEXT loop. We actually do have two good reasons. First, the idea of 1000 as a base may

or may not be correct. After you have the main program in, run it and listen to it. If it sounds too slow, simply change your base figure and everything will increase. Likewise, if the octave is too low, all you need to do is change the initial variables for each note. It's much easier than changing every CALL SOUND statement. Also, in spite of the fact that it may agree with the exact written sheet music, you may have to adjust certain portions, increasing or decreasing durations, until it just "sounds right".

One last word of warning before you spend a lot of time on this. One time or another, you have all probably seen the computer "pause" while scrolling up the screen or performing some other task. This is simply a function of the processor and occurs with greater regularity (and lasts longer) as you begin to fill up your memory. While in most programs this isn't really a problem, in the middle of a bar of music it can really drive you crazy. All we can recommend is that you start with a small song and then gradually and carefully move up to greater accomplishments.

```
*********************
*  HAPPY BIRTHDAY  *
*     V-PJ531KB     *
*    BY T CASTLE    *
*********************
```

DESCRIPTION. This is a novelty program, obviously suited to only one occasion. When the program is run, it first goes to an input section (subroutine 300-490) where it gets the child's age, name, and the speed at which you would like the "Happy Birthday" jingle played. While fast sounds a little better when played alone, we recommend medium for a "sing-a-long". After hitting any key the screen clears and "HAPPY BIRTHDAY" is displayed in double high, double wide letters. Again, after hitting a key, the words to the song "HAPPY BIRTHDAY" pop onto the screen and a white cake with the appropriate number of candles (from 1 to 9) appears to the right side of the screen.

Hitting a key "Lights" the candles and all candles begin to "flicker" in unison. Hitting another key plays the notes to the song, while the candles "flicker" in time to the music. A touch of a key returns the display to the large "HAPPY BIRTHDAY" display.

NOTES. The general sequence of the program is shown in lines 170 through 280. The subroutines which create the large letters are found in lines 800-1220 (creating the characters) and 2690-2910 (printing the letters). In Chapter 4 we discussed these in greater detail so a lengthy explanation will not be provided here.

This is a general purpose program which contains data statements in lines 2170 through 2590 to handle 1 through 9 candles. If you want to use it only once, you can shorten your input time by entering only the appropriate data statements for the child's age. For instance, line 2170 is one candle, 2200 is two, 2230 & 2240 is three, etc. You will also have to change line 1580. Instead of putting in ON AGE GOSUB ___, simply put in the appropriate GOSUB for that age (2170 for one, 2200 for two, etc).

We've been able to "pop" information to the screen, rather than have it scroll up or print character by character, through the use of three subroutines: 3020-3050 turns off all the characters; 3120-3180 turns on the cake and candles; and 3060-3110 turns on all of the other letters. This can be a useful technique when the step-by-step building process would detract from the overall appearance of the display.

Also note the subroutine at 3190-3280. This converts any letter Y (character code 89) to character code 64. Also note that in line 740 we redefined character 64 to the sixteen digit code for a "Y". Because of the number of characters we had to redefine for the cake and large letters, we needed to use set #8. By redefining character 89 we were able to have the letter "Y", but take it out of set #8.

```
100 REM   ******************
110 REM   * HAPPY BIRTHDAY *
120 REM   ******************
130 REM
140 REM   BY T CASTLE
150 REM   AMLIST V-PJ531KB
160 REM
170 REM    GET NAME & AGE
180 GOSUB 300
190 REM   SETS INIT VALUES
200 GOSUB 510
210 GOSUB 810
220 REM   START DISPLAY
230 GOSUB 2700
240 REM    BUILDS DISPLAY
250 GOSUB 1240
260 REM   PLAYS SONG
270 GOSUB 1780
280 GOTO 220
290 REM    GET NAME & AGE
300 CALL CLEAR
310 CALL SCREEN(12)
320 INPUT "CHILDS NAME?  ":N
M$
330 IF LEN(NM$)>9 THEN 320
340 GOSUB 3200
350 PRINT ::
360 INPUT "HOW OLD?      ":A
GE$
370 IF LEN(AGE$)<>1 THEN 360
380 AGE=ASC(AGE$)
390 IF (AGE<49)+(AGE>57)THEN
 360
400 PRINT ::
410 AGE=VAL(AGE$)
420 PRINT "ENTER S - SLOW"
430 PRINT "      M - MEDIUM"
440 PRINT "      F - FAST"::
450 INPUT "SPEED(S,M,F)? ":S
PEED$
460 IF (SPEED$="S")+(SPEED$=
"M")THEN 480
470 IF SPEED$="F" THEN 480 E
LSE 450
480 CALL CLEAR
490 RETURN
500 REM   SET VALUES
510 CALL SCREEN(12)
520 DIM MSG$(12)
530 MSG$(1)="HAPP@ BIRTHDA@"
540 MSG$(2)="  TO @OU --"
550 MSG$(3)="DEAR "
560 MSG$(4)="  TO @OU."
570 MSG$(5)=" HIT AN@ KE@ TO
"
580 MSG$(6)="LIGHT M@ CANDLE
S"
590 MSG$(7)="BLOW OUT CANDLE
S"
600 MSG$(8)="  DO  IT AGAIN"
610 MSG$(9)="   SING ALONG"
620 MSG$(10)=NM$
630 DATA FF7F3F1F0F070301,80
C0E0F0F8FCFEFF
640 DATA 7FBFDFEFF7FBFDFE,00
FFFFFFFFFFFFFF
650 DATA FFFFFFFFFFFFFFFF,FE
7E3E1E0E060200
660 DATA FEFEFEFEFEFEFEFE
670 RESTORE 630
680 FOR I=128 TO 134
690 READ A$
700 CALL CHAR(I,A$)
710 NEXT I
720 CALL CHAR(137,"070707070
7070707")
730 CALL CHAR(145,"070707070
7070707")
740 CALL CHAR(64,"0044442810
101010")
750 FLAME$(1)="0000040406030
702"
760 FLAME$(2)="0000010103060
702"
770 FLAME$(3)="0000000002020
202"
780 CALL CHAR(153,FLAME$(3))
790 RETURN
800 REM  INIT DATA BIG PRINT
810 DIM BG$(16)
820 DATA 0000,0303,0C0C,0F0F
,3030,3333,3C3C
830 DATA 3F3F,C0C0,C3C3,CCCC
,CFCF,F0F0,F3F3
840 DATA FCFC,FFFF
850 RESTORE 820
860 FOR I=1 TO 16
870 READ BG$(I)
880 NEXT I
890 REM  CHAR CODES FOR LTRS
```

```
900 DATA 004444447C444444,00
3844447C444444
910 DATA 0078444478404040,00
4444281010101010
920 DATA 0078242438242478,00
3810101010101038
930 DATA 0078444478504844,00
7C10101010101010
940 DATA 0078242424242478
950 SEQ$="0123340567819240"
960 RESTORE 890
970 FOR I=1 TO 9
980 READ LTR$(I)
990 J=87
1000 GOSUB 1030
1010 NEXT I
1020 RETURN
1030 FOR K=1 TO 16
1040 L$=SEG$(LTR$(I),K,1)
1050 L=ASC(L$)
1060 IF L<65 THEN 1090
1070 L=L-54
1080 GOTO 1100
1090 L=L-47
1100 L2=L2+1
1110 IF K>8 THEN 1140
1120 B$(L2)=B$(L2)&BG$(L)
1130 GOTO 1150
1140 B$(L2+2)=B$(L2+2)&BG$(L
)
1150 IF L2<2 THEN 1170
1160 L2=0
1170 NEXT K
1180 FOR M=1 TO 4
1190 CALL CHAR(M+J+((I*4)-4)
,B$(M))
1200 B$(M)=""
1210 NEXT M
1220 RETURN
1230 REM  BUILD DISPLAY
1240 CALL CLEAR
1250 GOSUB 3020
1260 PRINT " "&MSG$(1)
1270 PRINT " "&MSG$(2):::
1280 PRINT " "&MSG$(1)
1290 PRINT " "&MSG$(4):::
1300 PRINT " "&MSG$(1)
1310 PRINT " "&MSG$(3)&MSG$(
10):::
1320 PRINT " "&MSG$(1)
1330 PRINT " "&MSG$(4)::::
1340 GOSUB 3070
1350 DATA 17,19,18,20,19,21,
20,22
1360 DATA 13,28,14,29,15,30,
16,31
1370 DATA 13,19,14,20,15,21,
16,22
1380 RESTORE 1350
1390 FOR I=128 TO 130
1400 FOR K=1 TO 4
1410 READ A,B
1420 CALL HCHAR(A,B,I)
1430 NEXT K
1440 NEXT I
1450 FOR I=18 TO 20
1460 CALL HCHAR(I,23,132,9)
1470 NEXT I
1480 CALL HCHAR(17,23,131,9)
1490 FOR I=1 TO 4
1500 CALL HCHAR(12+I,19+I,13
2,8)
1510 NEXT I
1520 FOR I=1 TO 3
1530 CALL VCHAR(13+I,18+I,13
2,3)
1540 NEXT I
1550 CALL HCHAR(20,22,133)
1560 CALL VCHAR(17,22,134,3)
1570 GOSUB 3130
1580 ON AGE GOSUB 2170,2200,
2230,2270,2310,2360,2410,247
0,2530
1590 GOSUB 2610
1600 MS$="2307"&MSG$(5)
1610 GOSUB 2930
1620 MS$="2407"&MSG$(6)
1630 GOSUB 2930
1640 CALL KEY(3,KY,ST)
1650 IF ST=0 THEN 1640
1660 CALL CHAR(153,FLAME$(1)
)
1670 CALL HCHAR(24,7,32,20)
1680 MS$="2407"&MSG$(9)
1690 GOSUB 2930
1700 CALL CHAR(153,FLAME$(1)
)
1710 CALL KEY(3,KY,ST)
1720 CALL CHAR(153,FLAME$(2)
)
```

```
173Ø IF ST=Ø THEN 17ØØ
174Ø CALL HCHAR(23,7,32,2Ø)
175Ø CALL HCHAR(24,7,32,2Ø)
176Ø RETURN
177Ø REM  PLAY SONG
178Ø DATA 4ØØ,262,4ØØ,262,8Ø
Ø,294,8ØØ,262
179Ø DATA 8ØØ,349,12ØØ,33Ø,4
ØØ,44733
18ØØ DATA 4ØØ,262,4ØØ,262,8Ø
Ø,294,8ØØ,262
181Ø DATA 8ØØ,392,12ØØ,349,4
ØØ,44733
182Ø DATA 4ØØ,262,4ØØ,262,8Ø
Ø,523,8ØØ,44Ø
183Ø DATA 8ØØ,349,8ØØ,33Ø,8Ø
Ø,294,4ØØ,44733
184Ø DATA 4ØØ,466,4ØØ,466,8Ø
Ø,44Ø,8ØØ,349
185Ø DATA 8ØØ,392,12ØØ,349,4
ØØ,44733
186Ø RESTORE 178Ø
187Ø FOR I=1 TO 29
188Ø READ A,B
189Ø IF SPEED$="M" THEN 195Ø
19ØØ IF SPEED$="F" THEN 192Ø
191Ø IF SPEED$="S" THEN 194Ø
192Ø A=A*.8Ø
193Ø GOTO 195Ø
194Ø A=A*1.15
195Ø CALL SOUND(A,B,Ø)
196Ø IF TEST=1 THEN 2ØØØ
197Ø TEST=1
198Ø CALL CHAR(153,FLAME$(1)
)
199Ø GOTO 2Ø2Ø
2ØØØ CALL CHAR(153,FLAME$(2)
)
2Ø1Ø TEST=Ø
2Ø2Ø NEXT I
2Ø3Ø MS$="23Ø7"&MSG$(5)
2Ø4Ø GOSUB 293Ø
2Ø5Ø MS$="24Ø7"&MSG$(7)
2Ø6Ø GOSUB 293Ø
2Ø7Ø CALL CHAR(153,FLAME$(1)
)
2Ø8Ø CALL KEY(3,KY,ST)
2Ø9Ø CALL CHAR(153,FLAME$(2)
)
21ØØ IF ST=Ø THEN 2Ø7Ø
211Ø CALL CHAR(153,FLAME$(3)
)
212Ø CALL HCHAR(23,7,32,2Ø)
213Ø CALL HCHAR(24,7,32,2Ø)
214Ø CALL CLEAR
215Ø RETURN
216Ø REM  DATA FOR CANDLES
217Ø DATA 24,13,2,11,2,1Ø
218Ø RESTORE 217Ø
219Ø RETURN
22ØØ DATA 23,13,2,11,2,1Ø,26
,13,2,11,2,1Ø
221Ø RESTORE 22ØØ
222Ø RETURN
223Ø DATA 23,13,2,11,2,1Ø,26
,13,2,11,2,1Ø
224Ø DATA 25,13,3,12,1,11
225Ø RESTORE 223Ø
226Ø RETURN
227Ø DATA 22,13,1,1Ø,3,9,26,
13,1,1Ø,3,9
228Ø DATA 24,13,3,12,1,11,28
,13,3,12,1,11
229Ø RESTORE 227Ø
23ØØ RETURN
231Ø DATA 25,13,2,11,2,1Ø,22
,13,1,1Ø,3,9
232Ø DATA 26,13,1,1Ø,3,9,24,
13,3,12,1,11
233Ø DATA 28,13,3,12,1,11
234Ø RESTORE 231Ø
235Ø RETURN
236Ø DATA 22,13,1,1Ø,3,9,24,
13,1,1Ø,3,9
237Ø DATA 26,13,1,1Ø,3,9,23,
13,3,12,1,11
238Ø DATA 25,13,3,12,1,11,27
,13,3,12,1,11
239Ø RESTORE 236Ø
24ØØ RETURN
241Ø DATA 22,13,1,1Ø,3,9,26,
13,1,1Ø,3,9
242Ø DATA 23,13,2,11,2,1Ø,27
,13,2,11,2,1Ø
243Ø DATA 24,13,3,12,1,11,28
,13,3,12,1,11
244Ø DATA 25,13,2,11,2,1Ø
245Ø RESTORE 241Ø
```

```
2460 RETURN
2470 DATA 20,13,1,10,3,9,26,
13,1,10,3,9
2480 DATA 21,13,2,11,2,10,27
,13,2,11,2,10
2490 DATA 22,13,3,12,1,11,28
,13,3,12,1,11
2500 DATA 23,13,1,10,3,9,25,
13,3,12,1,11
2510 RESTORE 2470
2520 RETURN
2530 DATA 24,13,2,11,2,10,20
,13,1,10,3,9
2540 DATA 26,13,1,10,3,9,21,
13,2,11,2,10
2550 DATA 27,13,2,11,2,10,22
,13,3,12,1,11
2560 DATA 28,13,3,12,1,11,23
,13,1,10,3,9
2570 DATA 25,13,3,12,1,11
2580 RESTORE 2530
2590 RETURN
2600 REM  PRINT CANDLE
2610 FOR I=1 TO AGE
2620 READ CL1,RW1,RP1,RW2,RP
2,RW3
2630 CALL VCHAR(RW1,CL1,137,
RP1)
2640 CALL VCHAR(RW2,CL1,145,
RP2)
2650 CALL HCHAR(RW3,CL1,153)
2660 CALL SOUND(150,1600,0)
2670 NEXT I
2680 RETURN
2690 REM  PRINTS BIG
2700 RW=10
2710 CL=1
2720 GOSUB 3020
2730 ML=LEN(SEQ$)
2740 FOR I=1 TO ML
2750 NR=VAL(SEG$(SEQ$,I,1))
2760 IF NR=0 THEN 2840
2770 NR=(NR*4)+84
2780 GOTO 2800
2790 NR=NR+8
2800 CALL HCHAR(RW,CL,NR)
2810 CALL HCHAR(RW,CL+1,NR+1
)
2820 CALL HCHAR(RW+1,CL,NR+2
)
2830 CALL HCHAR(RW+1,CL+1,NR
+3)
2840 CL=CL+2
2850 NEXT I
2860 MS$="2409"&SEG$(MSG$(5)
,1,12)
2870 GOSUB 2930
2880 GOSUB 3070
2890 CALL KEY(3,KY,ST)
2900 IF ST=0 THEN 2890
2910 RETURN
2920 REM  PRINT MESSAGE
2930 MR=VAL(SEG$(MS$,1,2))
2940 MC=VAL(SEG$(MS$,3,2))
2950 LM=LEN(MS$)-4
2960 MG$=SEG$(MS$,5,LM)
2970 FOR I=1 TO LM
2980 CH=ASC(SEG$(MG$,I,1))
2990 CALL HCHAR(MR,MC+I,CH)
3000 NEXT I
3010 RETURN
3020 FOR I=1 TO 16
3030 CALL COLOR(I,1,1)
3040 NEXT I
3050 RETURN
3060 REM  TURN ON MAIN SETS
3070 CALL SOUND(200,1400,0)
3080 FOR I=1 TO 12
3090 CALL COLOR(I,2,1)
3100 NEXT I
3110 RETURN
3120 REM  ON-CAKE&CANDLE
3130 CALL SOUND(200,1400,0)
3140 CALL COLOR(13,16,1)
3150 CALL COLOR(14,6,16)
3160 CALL COLOR(15,6,1)
3170 CALL COLOR(16,9,1)
3180 RETURN
3190 REM  CONVERT Y'S
3200 NU$=""
3210 FOR I=1 TO LEN(NM$)
3220 T=ASC(SEG$(NM$,I,1))
3230 IF T<>89 THEN 3250
3240 T=64
3250 NU$=NU$&CHR$(T)
3260 NEXT I
3270 NM$=NU$
3280 RETURN
```

HAPPY COMPUTING!

```
*********************
*  MONKEY BUSINESS  *
*     V-PH431KB     *
*********************
```

DESCRIPTION. Here's a real learning tool for children under 6, which teaches both addition and subtraction with primary numbers from Ø-9. The fact that it's self pacing and offers tremendous instructor flexibility makes inputting well worth the effort for those who have the need. After the RUN command, the instructor sets up the program to fit the needs and abilities of the student. The options are thoroughly covered in lines 154Ø-239Ø of the printed program and display each time the program is RUN. After the instructor responses are obtained, the screen is cleared and the student display is presented.

The screen is a dark green background with three bands of medium green across the bottom representing ground level. To the right of the screen there's a black tree trunk with light green foliage at the top. Near the top of the trunk there are six bananas and at the base of the trunk there's a small yellow monkey. The questions appear on the remaining portions of the screen to the left. Each question is presented vertically in three different ways. To the far left, sets are presented with Ø-9 green dots in yellow squares. A separate block is presented for each number in the question and for the answer. The center and right columns show the same question in word form and numeric form as follows:

```
TWO          2
PLUS
ONE        + 1
           ---
THREE        3
```

The numeric answer is surrounded by a blinking red block. Level one shows: set theory, word problems; numeric problems; and answers. All the student has to do is match the number in the block to the number on the keyboard. At level two the numeric answer is not displayed. Level three shows only the word and numeric problem with no answers. At level four, only the numeric problem is displayed.

With each correct answer the monkey "squeaks" and climbs about half way up the tree. A wrong answer drops him down one notch, displays "wrong" at the bottom of the screen, and gives a small buzzer. If there are no wrong answers the monkey will reach the bananas with two correct answers and then he: retrieves one banana; slides down the tree; and stacks a banana to his right. Play continues in this manner until all six bananas are retrieved and then the student gets a "ting-a-ling" and six more bananas appear in the tree. After all possible questions have been presented at a given level, the student's progress is checked. If he answers 9Ø% correct the computer automatically moves up to the next level; otherwise, it repeats that level until 9Ø% is achieved. When the specified "end level" is reached or at the completion of level 4, a message appears at the bottom of the screen saying "YOU'RE THE TOP BANANA" and a suitable "ting-a-ling" is provided. At this point or at any other point during the session, by striking the letter "S", the instructor can ask for a report which shows; total number of questions, how many correct, how many wrong, and how many right or wrong, answers occurred with each of the numbers from Ø-9.

```
100 REM  ******************
110 REM  * MONKEY BUSINESS *
120 REM  ******************
130 REM
140 REM  BY T CASTLE
150 REM  AMLIST V-PH431KB
160 REM
170 GOSUB 1380
180 GOSUB 300
190 GOSUB 560
200 GOSUB 3160
210 IF KY=83 THEN 270
220 GOSUB 4260
230 IF LEV<ENDLEV THEN 200
240 GOSUB 4580
250 CALL KEY(3,KY,ST)
260 IF ST=0 THEN 250
270 GOSUB 4790
280 GOTO 170
290 REM  SET INIT VARIABLES
300 CALL CLEAR
310 CALL SCREEN(13)
320 DATA 1,13,13,11,3,1,2
330 DATA 1,11,1,11,2,4,13
340 RESTORE 320
350 FOR I=10 TO 16
360 READ A,B
370 CALL COLOR(I,A,B)
380 NEXT I
390 DATA 104,000000FFFFFF,10
5,0000FFFFFF
400 DATA 106,1C1C1C1C1C1C1C1
C,107,3838383838383838
410 DATA 108,0000001F1F1F1C1
C,109,000000F8F8F83838
420 DATA 110,1C1C1F1F1F,111,
3838F8F8F8
430 DATA 113,00003C3C3C3C,12
0,FFFFFFFFFFFFFFFF
440 DATA 128,FFFFFFFFFFFFFFF
F,136,1C3E3E3E1CFEFFFF
450 DATA 137,7F7F7F7F3CFCF0C
0,138,0
460 DATA 139,183C246666C3C38
1,140,103020606C0C080
470 DATA 144,00000000000007070
7,112,0
480 DATA 145,183C246666C3C38
1,146,103020606C0C080
490 DATA 147,080C0406060303O
1,152,FFFFFFFFFFFFFFFF
500 FOR I=1 TO 22
510 READ A,A$
520 CALL CHAR(A,A$)
530 NEXT I
540 RETURN
550 REM  START DISPLAY
560 CALL VCHAR(7,24,128,15)
570 CALL VCHAR(4,24,145,3)
580 DATA 1,23,152,3,2,22,152
,5,3,22,152,5
590 DATA 4,21,152,3,4,25,152
,3,5,20,152,3
600 DATA 5,26,152,3,6,21,152
,2,6,21,152,2
610 DATA 6,26,152,2,7,22,152
,1,7,26,152,1
620 DATA 23,1,120,32,24,1,12
0,32,22,1,120
630 DATA 32,14,14,104,1,15,1
3,106,1,15,15
640 DATA 107,1,16,14,105,1,1
4,13,108,1,14
650 DATA 15,109,1,16,13,110,
1,16,15,111,1
660 FOR I=1 TO 23
670 READ A,B,C,D
680 CALL HCHAR(A,B,C,D)
690 NEXT I
700 I=21
710 GOSUB 740
720 RETURN
730 REM   MOVES UP TREE
740 IF I+1>21 THEN 780
750 CALL SOUND(30,1500,5)
760 CALL HCHAR(I+1,24,128)
770 CALL HCHAR(I+1,25,138)
780 CALL HCHAR(I-1,24,144)
790 CALL HCHAR(I,24,144)
800 CALL HCHAR(I-1,25,136)
810 CALL HCHAR(I,25,137)
820 CALL SOUND(30,1500,5)
830 RETURN
840 REM  MOVES DOWN TREE
850 IF I+1>21 THEN 940
860 CALL SOUND(30,1500,5)
870 CALL HCHAR(I-1,24,128)
```

```
88Ø CALL HCHAR(I-1,25,138)
89Ø CALL HCHAR(I,24,144)
9ØØ CALL HCHAR(I+1,24,144)
91Ø CALL HCHAR(I,25,136)
92Ø CALL HCHAR(I+1,25,137)
93Ø CALL SOUND(3Ø,15ØØ,5)
94Ø RETURN
95Ø REM  SLIDE DOWN TREE
96Ø GTB=6
97Ø CALL GCHAR(GTB,24,FND)
98Ø IF FND=145 THEN 99Ø ELSE
 1Ø1Ø
99Ø CALL HCHAR(GTB,24,146)
1ØØØ GOTO 1Ø8Ø
1Ø1Ø IF FND=146 THEN 1Ø2Ø EL
SE 1Ø4Ø
1Ø2Ø CALL HCHAR(GTB,24,128)
1Ø3Ø GOTO 1Ø8Ø
1Ø4Ø GTB=GTB-1
1Ø5Ø UP1=UP1-1
1Ø6Ø GOSUB 76Ø
1Ø7Ø GOTO 97Ø
1Ø8Ø FOR I=UP1 TO 2Ø
1Ø9Ø GOSUB 87Ø
11ØØ NEXT I
111Ø UP1=21
112Ø GTB=28
113Ø CALL GCHAR(21,GTB,FND)
114Ø IF FND=32 THEN 115Ø ELS
E 117Ø
115Ø CALL HCHAR(21,GTB,14Ø)
116Ø GOTO 136Ø
117Ø CALL GCHAR(21,GTB,FND)
118Ø IF FND=14Ø THEN 119Ø EL
SE 122Ø
119Ø CALL HCHAR(21,GTB,139)
12ØØ IF GTB=3Ø THEN 124Ø
121Ø GOTO 136Ø
122Ø GTB=GTB+1
123Ø GOTO 113Ø
124Ø FOR I=1 TO 1Ø
125Ø CALL SOUND(1Ø,9ØØ,Ø)
126Ø CALL SOUND(1Ø,6ØØ,Ø)
127Ø NEXT I
128Ø FOR I=2 TO 15
129Ø CALL GCHAR(I,3Ø,XR)
13ØØ IF XR<>32 THEN 133Ø
131Ø CALL HCHAR(I,3Ø,139,3)
132Ø I=15
133Ø NEXT I
134Ø CALL VCHAR(4,24,145,3)
135Ø CALL HCHAR(21,28,32,3)
136Ø RETURN
137Ø REM  GETS TEACHER INST
138Ø CALL SCREEN(4)
139Ø CALL CLEAR
14ØØ PRINT TAB(5);"TEACHER I
NSTRUCTIONS"::
141Ø PRINT ::"MONKEY BUSINES
S IS A LEARN-"
142Ø PRINT "ING  PROGRAM  FO
R  TEACHING"
143Ø PRINT "PRIMARY NUMBERS
FROM Ø - 9."::
144Ø PRINT :"IT TEACHES BOTH
 ADDITION AND"
145Ø PRINT "SUBTRACTION AT T
HE CHOICE OF"
146Ø PRINT "THE INSTRUCTOR."
::::::::
147Ø PRINT "ENTER A FOR ADDI
TION"
148Ø INPUT "       S FOR SUBT
RACTION? ":Q$
149Ø IF Q$="A" THEN 151Ø
15ØØ IF Q$="S" THEN 153Ø ELS
E 147Ø
151Ø TYP=1
152Ø GOTO 154Ø
153Ø TYP=2
154Ø CALL CLEAR
155Ø PRINT "THE INSTRUCTOR M
AY SPECIFY"
156Ø PRINT "THE  MAXIMUM  AN
D  MINIMUM"
157Ø PRINT "NUMBER TO BE USE
D IN  BOTH"
158Ø PRINT "THE QUESTION AND
 ANSWER."::::
159Ø PRINT "NUMBERS FROM Ø T
O 9 MAY BE"
16ØØ PRINT "SPECIFIED."::::
161Ø PRINT "IF THE INSTRUCTO
R DOES NOT"
162Ø PRINT "WANT TO SET VALU
ES, Ø TO 9"
```

```
1630 PRINT "WILL BE USED."::
::::
1640 PRINT "ENTER S TO SET V
ALUES"
1650 INPUT "        D TO DEFAU
LT?   ":Q$
1660 IF Q$="D" THEN 1860
1670 IF Q$="S" THEN 1680 ELS
E 1640
1680 CALL CLEAR
1690 INPUT "MAX IN ANSWER?
 ":Q$
1700 IF LEN(Q$)>1 THEN 1690
1710 IF (ASC(Q$)<48)+(ASC(Q$
)>57)THEN 1690
1720 MXAN=VAL(Q$)
1730 INPUT "MIN IN ANSWER?
 ":Q$
1740 IF LEN(Q$)>1 THEN 1730
1750 IF (ASC(Q$)<48)+(ASC(Q$
)>57)THEN 1730
1760 MNAN=VAL(Q$)
1770 INPUT "MAX IN QUESTION?
 ":Q$
1780 IF LEN(Q$)>1 THEN 1770
1790 IF (ASC(Q$)<48)+(ASC(Q$
)>57)THEN 1770
1800 MXNR=VAL(Q$)
1810 INPUT "MIN IN QUESTION?
 ":Q$
1820 IF LEN(Q$)>1 THEN 1810
1830 IF (ASC(Q$)<48)+(ASC(Q$
)>57)THEN 1810
1840 MNNR=VAL(Q$)
1850 GOTO 1900
1860 MXNR=9
1870 MNNR=0
1880 MXAN=9
1890 MNAN=0
1900 CALL CLEAR
1910 PRINT "INSTRUCTOR MAY
SPECIFY  THE"
1920 PRINT "ORDER IN WHICH T
HE QUESTIONS"
1930 PRINT "WILL BE PRESENTE
D."::
1940 PRINT "1 - WILL GIVE  Q
UESTIONS  IN"
1950 PRINT "SEQUENTIAL ORDER
 WITHIN  THE"
1960 PRINT "SPECIFIED RANGE.
"::
1970 PRINT "2 - GIVES  RANDO
M  QUESTIONS"
1980 PRINT "WITHIN THE RANGE
, USING EACH"
1990 PRINT "NUMBER IN EACH P
OSITION."::
2000 PRINT "3 - GIVES  SEQUE
NTIAL  ORDER"
2010 PRINT "FOLLOWED BY RAND
OM ORDER FOR"
2020 PRINT "EACH LEVEL.":::::
::::
2030 INPUT "ENTER 1, 2 OR 3?
 ":Q$
2040 IF LEN(Q$)>1 THEN 2030
2050 IF (ASC(Q$)<49)+(ASC(Q$
)>51)THEN 2030
2060 ORD=VAL(Q$)
2070 GOSUB 4330
2080 CALL CLEAR
2090 PRINT "THERE ARE FOUR L
EVELS OF USE"
2100 PRINT "FOR THIS PROGRAM
."::
2110 PRINT "LEV 1- DISPLAYS
SETS, WORDS,"
2120 PRINT "NUMERIC QUESTION
 AND ANSWER."::
2130 PRINT "LEV 2- SAME  DIS
PLAY WITHOUT"
2140 PRINT "NUMERIC ANSWER."
::
2150 PRINT "LEV 3- DISPLAYS
NUMERIC  AND"
2160 PRINT "WORD QUESTION ON
LY."::
2170 PRINT "LEV 4- NUMERIC Q
UESTION ONLY"
2180 PRINT ::::::::
2190 INPUT "START AT 1 TO 4?
 ":Q$
2200 IF LEN(Q$)>1 THEN 2190
2210 IF (ASC(Q$)<49)+(ASC(Q$
)>52)THEN 2190
```

```
2220 LEV=VAL(Q$)
2230 STLEV=LEV
2240 INPUT "END AT   1 TO 4?
 ":Q$
2250 IF LEN(Q$)>1 THEN 2240
2260 IF (ASC(Q$)<49)+(ASC(Q$
)>52)THEN 2240
2270 ENDLEV=VAL(Q$)+1
2280 UP3=7
2290 UP1=21
2300 UP2=7
2310 QRT=0
2320 QWR=0
2330 FOR I=0 TO 9
2340 AWR(I)=0
2350 ART(I)=0
2360 NEXT I
2370 CHKR=0
2380 CHKW=0
2390 RETURN
2400 REM  BLOCK DISPLAYS
2410 NR=VAL(SEG$(MSG$,5,1))+
1
2420 IF NR>5 THEN 2450
2430 ON NR GOSUB 2560,2580,2
600,2620,2640
2440 GOTO 2470
2450 NR=NR-5
2460 ON NR GOSUB 2660,2680,2
700,2720,2740
2470 FOR PR=1 TO 3
2480 X=VAL(SEG$(MSG1$,6+PR,1
))
2490 A=VAL(SEG$(MSG1$,10,2))
-1+PR
2500 B=VAL(SEG$(MSG1$,12,2))
2510 IF LEV>2 THEN 2530
2520 ON X GOSUB 2760,2780,28
10,2850
2530 NEXT PR
2540 GOSUB 2890
2550 GOTO 2870
2560 MSG1$="ZERO 0111"&MSG$
2570 RETURN
2580 MSG1$="ONE  1121"&MSG$
2590 RETURN
2600 MSG1$="TWO  2131"&MSG$
2610 RETURN
2620 MSG1$="THREE3141"&MSG$
2630 RETURN
2640 MSG1$="FOUR 4313"&MSG$
2650 RETURN
2660 MSG1$="FIVE 5323"&MSG$
2670 RETURN
2680 MSG1$="SIX  6414"&MSG$
2690 RETURN
2700 MSG1$="SEVEN7424"&MSG$
2710 RETURN
2720 MSG1$="EIGHT8434"&MSG$
2730 RETURN
2740 MSG1$="NINE 9444"&MSG$
2750 RETURN
2760 CALL HCHAR(A,B,112,3)
2770 RETURN
2780 CALL HCHAR(A,B,112,3)
2790 CALL HCHAR(A,B+1,113)
2800 RETURN
2810 CALL HCHAR(A,B,112,3)
2820 CALL HCHAR(A,B,113)
2830 CALL HCHAR(A,B+2,113)
2840 RETURN
2850 CALL HCHAR(A,B,113,3)
2860 RETURN
2870 RETURN
2880 REM  PRINTS NUMB MSG
2890 IF CNT=1 THEN 2910
2900 IF LEV>1 THEN 2930
2910 XL=ASC(SEG$(MSG1$,6,1))
2920 CALL HCHAR(A-1,B+12,XL)
2930 IF LEV>3 THEN 3000
2940 IF LEV<3 THEN 2960
2950 IF CNT=0 THEN 3000
2960 FOR LP2=1 TO 5
2970 XL=ASC(SEG$(MSG1$,LP2,1
))
2980 CALL HCHAR(A-1,B+3+LP2,
XL)
2990 NEXT LP2
3000 RETURN
3010 REM  PRINT SIGNS & LINE
3020 IF TYP=1 THEN 3030 ELSE
 3050
3030 SYM$="PLUS +"
3040 GOTO 3060
3050 SYM$="MINUS-"
3060 IF LEV>3 THEN 3110
```

```
3070 FOR SYM=1 TO 5
3080 XL=ASC(SEG$(SYM$,SYM,1)
)
3090 CALL HCHAR(8,SYM+5,XL)
3100 NEXT SYM
3110 CALL HCHAR(11,13,95,3)
3120 XL=ASC(SEG$(SYM$,6,1))
3130 CALL HCHAR(10,13,XL)
3140 RETURN
3150 REM  MAIN CONTROL LOOP
3160 IF ORD=2 THEN 3230
3170 FOR I=1 TO J
3180 ORD1(I)=SETQ1(I)
3190 ORD2(I)=SETQ1(I)
3200 NEXT I
3210 PASS=PASS+1
3220 IF PASS=1 THEN 3270
3230 FOR I=1 TO J
3240 ORD1(I)=SETQ2(I)
3250 ORD2(I)=SETQ3(I)
3260 NEXT I
3270 FOR HH=1 TO J
3280 FOR LL=1 TO J
3290 GOSUB 4170
3300 IF TYP=1 THEN 3340
3310 IF ORD1(HH)-ORD2(LL)<MN
AN THEN 3650
3320 IF ORD1(HH)-ORD2(LL)>MX
AN THEN 3650
3330 GOTO 3360
3340 IF ORD1(HH)+ORD2(LL)>MX
AN THEN 3650
3350 IF ORD1(HH)+ORD2(LL)<MN
AN THEN 3650
3360 MSG$="0402"&STR$(ORD1(H
H))
3370 CNT=1
3380 HLDN(1)=ORD1(HH)
3390 GOSUB 2410
3400 GOSUB 3020
3410 MSG$="0902"&STR$(ORD2(L
L))
3420 HLDN(2)=ORD2(LL)
3430 GOSUB 2410
3440 CNT=0
3450 IF TYP=1 THEN 3480
3460 NR=ORD1(HH)-ORD2(LL)
3470 GOTO 3490
3480 NR=ORD1(HH)+ORD2(LL)
3490 MSG$="1402"&STR$(NR)
3500 HLDN(3)=NR
3510 GOSUB 2410
3520 IF LEV>1 THEN 3540
3530 GOTO 3550
3540 IF LEV>2 THEN 3550
3550 CALL COLOR(10,7,13)
3560 CALL KEY(3,KY,ST)
3570 IF KY=83 THEN 3740
3580 CALL COLOR(10,1,13)
3590 IF ST=0 THEN 3550
3600 IF KY<>ASC(SEG$(MSG1$,6
,1))THEN 3630
3610 GOSUB 3760
3620 GOTO 3650
3630 GOSUB 3960
3640 GOTO 3550
3650 NEXT LL
3660 NEXT HH
3670 IF ORD<3 THEN 3740
3680 IF PASS=2 THEN 3740
3690 IF CHKR/(CHKW+CHKR)>.90
 THEN 3160
3700 PASS=0
3710 CHKR=0
3720 CHKW=0
3730 GOTO 3160
3740 RETURN
3750 REM  CORRECT RESPONSE
3760 CHKR=CHKR+1
3770 QRT=QRT+1
3780 ART(HLDN(1))=ART(HLDN(1
))+1
3790 ART(HLDN(2))=ART(HLDN(2
))+1
3800 ART(HLDN(3))=ART(HLDN(3
))+1
3810 GOSUB 2890
3820 CALL HCHAR(15,14,KY)
3830 IF UP1-UP3>7 THEN 3900
3840 FOR I=UP1 TO 8 STEP -1
3850 GOSUB 740
3860 NEXT I
3870 UP1=8
3880 GOSUB 960
3890 GOTO 3940
3900 FOR I=UP1 TO UP1-UP3 ST
EP -1
```

```
3910 GOSUB 740
3920 NEXT I
3930 UP1=UP1-UP3
3940 RETURN
3950 REM  WRONG RESPONSE
3960 MSG3$="WRONG"
3970 QWR=QWR+1
3980 AWR(HLDN(1))=AWR(HLDN(1
))+1
3990 AWR(HLDN(2))=AWR(HLDN(2
))+1
4000 AWR(HLDN(3))=AWR(HLDN(3
))+1
4010 FOR I=1 TO 5
4020 XL=ASC(SEG$(MSG3$,I,1))
4030 CALL HCHAR(23,I+13,XL)
4040 NEXT I
4050 CALL SOUND(300,110,0)
4060 CHKW=CHKW+1
4070 FOR I=UP1 TO UP1
4080 IF I+1>21 THEN 4100
4090 GOSUB 850
4100 NEXT I
4110 UP1=UP1+1
4120 IF UP1>21 THEN 4130 ELS
E 4140
4130 UP1=21
4140 CALL HCHAR(23,13,120,6)
4150 RETURN
4160 REM  ERASE ROUTINE
4170 FOR I=3 TO 13
4180 CALL HCHAR(I,2,32,15)
4190 NEXT I
4200 CALL HCHAR(14,2,32,11)
4210 CALL HCHAR(15,2,32,11)
4220 CALL HCHAR(16,2,32,11)
4230 CALL HCHAR(15,14,32)
4240 RETURN
4250 REM  CHECK & INCR LEVEL
4260 IF CHKR/(CHKW+CHKR)<.90
 THEN 4280
4270 LEV=LEV+1
4280 PASS=0
4290 CHKR=0
4300 CHKW=0
4310 RETURN
4320 REM  CREATES SETS
4330 RANDOMIZE
4340 J=MXNR-MNNR+1
4350 FOR I=1 TO J
4360 SETQ1(I)=MNNR+I-1
4370 SETQ2(I)=MNNR+I-1
4380 SETQ3(I)=MNNR+I-1
4390 NEXT I
4400 FOR I=1 TO J
4410 T1=INT((J-1+1)*RND)+1
4420 RM1=SETQ2(T1)
4430 T2=INT((J-1+1)*RND)+1
4440 IF T2=T1 THEN 4430
4450 RM2=SETQ2(T2)
4460 SETQ2(T1)=RM2
4470 SETQ2(T2)=RM1
4480 T1=INT((J-1+1)*RND)+1
4490 RM1=SETQ3(T1)
4500 T2=INT((J-1+1)*RND)+1
4510 IF T2=T1 THEN 4500
4520 RM2=SETQ3(T2)
4530 SETQ3(T1)=RM2
4540 SETQ3(T2)=RM1
4550 NEXT I
4560 RETURN
4570 REM  END OF RUN
4580 MSG3$="YOU'RE THE TOP B
ANANA"
4590 FOR I=1 TO LEN(MSG3$)
4600 XL=ASC(SEG$(MSG3$,I,1))
4610 CALL HCHAR(22,I+5,XL)
4620 NEXT I
4630 FOR I=1 TO 30
4640 CALL SOUND(40,900,0)
4650 CALL SOUND(40,500,0)
4660 NEXT I
4670 MSG3$="CALL INSTRUCTOR"
4680 FOR I=1 TO LEN(MSG3$)
4690 XL=ASC(SEG$(MSG3$,I,1))
4700 CALL HCHAR(23,I+8,XL)
4710 NEXT I
4720 MSG3$="  HIT ANY KEY  "
4730 FOR I=1 TO LEN(MSG3$)
4740 XL=ASC(SEG$(MSG3$,I,1))
4750 CALL HCHAR(24,I+8,XL)
4760 NEXT I
4770 RETURN
4780 REM  INSTRUCTOR REPORT
4790 CALL CLEAR
4800 CALL SCREEN(4)
```

```
4810 IF TYP=1 THEN 4840
4820 TYP$="SUBTRACTION"
4830 GOTO 4850
4840 TYP$="ADDITION"
4850 PRINT "TYPE   - ";TYP$
4860 PRINT "LEVELS - ";STLEV
;" TO";ENDLEV-1
4870 PRINT
4880 PRINT "NUMBER   WRONG
 RIGHT"::
4890 FOR I=0 TO 9
4900 PRINT I;TAB(12);AWR(I);
TAB(19);ART(I)
4910 NEXT I
4920 PRINT :::
4930 PRINT "TOTAL QUESTIONS
- ";QWR+QRT
4940 PRINT "TOTAL CORRECT
- ";QRT
4950 PRINT "TOTAL WRONG
- ";QWR
4960 PER=INT((QRT/(QWR+QRT))
*100+.5)
4970 PRINT "PERCENTAGE
- ";PER;"%"
4980 PRINT "HIT ANY KEY TO C
ONTINUE"
4990 CALL KEY(3,KY,ST)
5000 IF ST=0 THEN 4990
5010 RETURN
```

CHAPTER SIX

Data Files

GENERAL. Nothing probably causes more confusion for the new computer owner than the subject of data and data files. By the same token, a clear understanding of data and data files is absolutely essential for almost all advanced uses of any computer system. Part of this confusion arises from the repetitive use of the word "data", which so often crops up when one tries to discuss any aspect of data processing (there's that word again).

To begin this chapter we're going to discuss the different types of data so that we can readily distinguish the difference between data files and all other types of data. Next, we're going to discuss the practical uses of these files for someone limited to console basic and one recorder. Given these limitations we're going to show you what we believe is the most efficient way to transfer information into and out of data files. Finally, although none of the programs in this manual will require it, we feel obligated to point out some of the far greater capabilities of data files if you expand your system to disk drive or simply an additional recorder.

Data. For our purposes, let's define data as being "anything to which a variable name could be assigned". A line number, for instance, is not data.

```
>100 A=130
>110 GOTO A
>120 STOP
>130 PRINT "I'M HERE"
```

Running the above program produces the error message "INCORRECT STATEMENT IN 110". Line numbers, commands, and words such as REM are not items of data, but are parts of a program. Data is information which can be stored, changed, and/or manipulated in some way. It is, to a computer program, what gasoline is to an automobile; or what electricity is to a television. A computer program generally gets its data from one of five sources:

First, the program can store some data as part of a data statement, such as:

```
>100 DATA JOHN,10
>110 READ A$,A
```

The word "JOHN" and the number "10" are both elements of data and, by using the READ statement, we can make A$="JOHN" and A=10.

Second, we can create data directly in a program with a statement such as:

```
>100 J=20
```

Third, the computer can generate some data of its own by using other data already provided.

```
>100 A=10
>120 B=20
>130 C=A+B
```

In the above example, the value of 30, an element of data, was generated by the computer through a formula.

Fourth, we can acquire data through the use of an INPUT statement such as:

```
>100 INPUT "LAST NAME? ":A$
```

In this case, the computer pauses and the user is asked to enter some "data".

The fifth method is similar to the INPUT statement above, in that the computer is looking to something "outside" of the computer program to provide the data. Assuming a file has been opened and information has previously been stored, the input statement might look like one of these:

```
>100 INPUT #1:X$
>110 INPUT #2:A,B,C
```

It may appear from the above discussion that "data is data" and there's no significant difference between putting information directly into a program or getting it from some other source; however, there is one advantage that only "data files" can provide. Data files are the only method, short of changing the actual basic program, by which you can save the values of each variable, string or numeric, so that they can be recovered at a later time. You could write a program which permitted you to enter (input from the screen) all of your golf scores for the past six weeks; you could create variables to equal each of these and more variables to give you handicap and average; and you could then display all of this information on the screen. However, once you shut off the computer, unless this information is saved in "data files", it is lost. If you played one more round of golf, you would have to reenter all of the previous data again to arrive at your new totals. With data files, you simply READ in the previous data, input the next score, and then display the new totals.

Creating Files. We're going to discuss the advantages and differences involved in using multiple recorders and/or disk drive later (and there are many); however the following discussion is limited to use of one cassette recorder only and straight console basic. In order to demonstrate the principles and limitations involved, we want you to envision a 3 X 5 card file (just the plain metal box). This box represents your cassette recorder (and cassette). It's capable of storing information; however, without cards and without information on those cards, the box can tell you nothing. In order to be useful, information must be entered on 3 X 5 cards and those cards need to be filed in an orderly fashion. Suppose you wanted to keep track of all furniture and appliances that you purchase (with a value in excess of $100.00) and, for insurance purposes, you want to know what the item was, the make, the model, the price and when it was purchased. To begin building a file, we would want to first get an empty box and a blank stack of cards. In computer terms this means that we would open a file using a statement similar to this:

```
>100 OPEN #1:"CS1",INTERNAL,O
 UTPUT,FIXED 128
```

Next, having opened our box, we might start by developing a card that looks like this:

```
| Television - Panasonic   MTV1876  |
|              032283      456.00   |
|                                   |
|                                   |
|                                   |
|                                   |
```

If you had a second television, since there's room left on the card, you could just add it to the same card. Your finished card would look like this:

```
| Television - Panasonic   MTV1876  |
|              Ø32283      456.ØØ   |
|                                   |
|              Zenith      JJ123    |
|              Ø62Ø79      158.ØØ   |
|___________________________________|
```

To create a "card" on the computer is quite different than writing it on a card. There are several ways to record information on data files, some of which will be touched on later; however, the procedure we normally use, and the one we feel is the most efficient for console basic, might best be described as "a place for everything and everything in its place". Using this method we decide, in advance, in exactly what position in each record each piece of information will be found. Whether you're going to input the attached "Bowling Stats" program or not, we want you to study the subroutines from 342Ø through 4Ø3Ø. What we have done there is set up an array called NM$(X) which holds the names of up to six players. Comparing this to the household inventory program that we have been discussing, it would be similar to setting up an array with names like: "Television", "Beds", "Chairs", etc. Then we decide what kind of information and how many groups of information we'll need for each one of these categories. In the bowling program we needed to record data for at least 38 bowling series and, for each series, we needed to know the score of each of 3 games, how many times they had a chance to win a "kitty", and how many times they actually did win it. In the household program we're only recording two groups of information for each category. Each of these groups will contain the make, model, date purchased, and price. It should be noted that for each category you could reserve room for 3, 5, or many more pieces under each category. Our choice of two is purely arbitrary.

We'll cover this point more fully later, but accept for now that the most efficient way to use cassette data files is to work with maximum length records. Refer to your URG and you'll see that we have a choice between 64, 128, and 192 characters per line. In order to establish what our record will look like we set up "blanks" for each grouping of data. In the bowling program, a blank for each series would look like this - " Ø Ø ØØØ". The first three digits would be the score for game 1; the second 3 (position 4-6) would be the second game; positions 7-9 would be the third game; position 1Ø is the number of tries; and position 11 indicates number of wins. Taking 192 and dividing it by 11 spaces tells us that we could get a full 17 groups of data on each 192 character record, and that we're going to need 3 lines (records) of data to record the information for 38 series. Putting this all in perspective, we arrived at a plan which allowed 3 records for each bowler or 18 total records for the entire team (6 X 3). The first record for each bowler has the bowlers name in positions 1-11, followed by 12 groupings of 11 characters representing 12 of the 38 series. Each of the next two records contains 13 groupings of 11 characters. This arrangement is very easy for the computer to handle since it can easily find the name of each bowler in SEG$(X$,1,11) of record 1, 4, 7, 1Ø,

13, and 16. If you ask it for a particular series, it can easily calculate exactly at what position it would be in and in which record. Taking information from our household inventory card to prepare a data record, we might allow: 10 spaces for the type of appliance or furniture; 10 spaces for the make; 10 spaces for the model; 6 spaces for date; and 7 spaces for the amount. Our comparable record would look like this (spaced for the normal screen):

```
>110 RECORD$="TELEVISIONPANAS
 ONIC MTV1876    032283 456.00
 ZENITH    JJ123      062079 1
 58.00"
```

Of course this is somewhat smaller (76 characters) and it only allows for two extra groups of data instead of the 38 used in the bowling program. Each application requires its own analysis.

Going back to our card file analogy, once the card was completed you could store it immediately in the 3 X 5 metal box; likewise, once a data record is completed it can be printed to a data file. However, the computer does have certain rules which must be obeyed regarding what is sent to the file and the order of filing records. One rule is that the the computer would add enough blank spaces to your record to make it either 64, 128, or 192 characters long (notice we specified 128 in our opening statement). Another rule is that, on a cassette recorder, all records are entered sequentially. Applying this to our card file, what this means is that each card that is completed must be stored directly behind the previous card. If you would prefer that cards be filed alphabetically by make, by dollar value, or by date of purchase, you'll have to keep the cards out of the box (or the information out of the data file) until all are completed. As far as the computer is concerned, this means that all information, for all records, must be held within the available memory of the CPU prior to transfering any of it to the data file. Regardless of when you do it, at each point you want to send a record to the file, the entry would be similiar to the following:

```
>120 PRINT #1:RECORD$
```

Let's assume now that you've completed 30 cards and that you've stored them in the 3 X 5 file, either one at a time, as they were completed, or after sorting them. Let's discuss what you could do with them, how you could add new items, and how you could delete or remove old cards.

Reading from Data Files. Once created, one of the more common uses for data files is to simply read from them and compile certain information from them. For instance, in our card file, we might want to total up the value of all of the televisions that have been recorded. The computer method of performing this task is very meticulous and methodic. The process involves: opening the file; getting a copy of each record; analyzing the copy; throwing the copy away; getting the next record; and repeating the process until all cards have been analyzed. Following is a sample program that demonstrates this process:

```
>100 OPEN #1:"CS1",INTERNAL,I
 NPUT ,FIXED 128
>110 FOR I=1 TO 30
>120 INPUT #1:RECORD$
>130 IF SEG$(RECORD$,1,10)="T
 ELEVISION" THEN 140 ELSE 180
>140 FOR J=37 TO 70 STEP 23
```

```
>150 IF SEG$(RECORD$,J,7)="
      " THEN 170
>160 TOTAL=TOTAL+VAL(SEG$(REC
 ORD$,J,7))
>170 NEXT J
>180 NEXT I
>190 CLOSE #1
>200 PRINT "TOTAL =";TOTAL
```

The above program is a simple "search" program. We open our file in the INPUT mode, using the same specifications used when we built the file. Since we know that we have 30 records to search, we simply set up a loop (FOR I=1 TO 30) and input each of 30 records, using the SEG$ command to check for what type of furniture or appliance that record contains. If it is not the type we want, we cause the program to go to the NEXT I (next record). If it does find the appropriate card, we set up an additional loop (FOR J=37 TO 70 STEP 23) to look at particular sections of that data record and compile the information found there. Notice how we've used the FOR statement to specify the position in the data record. There are only two groupings of information in each record and it would be possible to use a statement such as FOR J=1 TO 2 instead of 37-70; however, we would then have to specify the position for each grouping in separate statements.

In the above application, at the completion of the search, we had a value for TOTAL; however, all of the individual data on each card was still on cassette and not in the CPU (in the memory of the computer). A more common application of this type of loop is found in lines 430-640 of the bowling program. Analyze this closely and you'll understand how we loaded information for 6 players (FOR I=1 to 6, line 510) and three lines per player (FOR AD=0 TO 26 STEP 13, line 520). All of the information found, rather than being compiled, was brought into memory and stored in ARRAYS. NM$(I) stored the name of each player and BW$(I,K) stored the 11 digit series information for each player. Once in memory, we can manipulate and reference this information in a number of ways by specifying the player, series, and what type of information we want without having to go back to the cassette again. For instance, the value of the score of the first game bowled, in the 19th series, for player 4 can by found through use of the following statement:

```
>100 PRINT VAL(SEG$(BW$(4,19),1,3))
```

Why is it necessary to pull all this data into memory? The answer is that "it isn't", if all you need is one particular piece of information or one set of figures; however, each time you want any information from the file you will have to begin at record 1 and input each record sequentially. In the bowling program, with 18 records, this means it will take about 2 minutes to search each time it needs some information. By storing it in memory, the item can be referenced directly.

Adding/Deleting/Changing. We talked above about bringing entire files into memory and storing information in arrays. When operating with a single recorder and console basic, this procedure is essential. Without it, adding, deleting and changing data records is an impossible task. Previously written checks may need to be corrected; bowling scores need to be updated or added to each week; new invoices are issued periodically; and names need to be added to or deleted

from files. We're going to fall back on our 3 X 5 box again to explain this principle.

Assume you have previously recorded information on your cards and that you have 25 cards in the box. When you open the box we're going to permit you to do only one of two things: first, you can take all of the cards out and hold them in your hand or; second, you can throw away all the cards and create a whole new stack before putting them back in. You cannot fill out a new card and stick it in the box behind all of the others; you cannot reach over to the box and pull out card 15 and throw it away; and you cannot remove a card, change it, and put it back. Further, if you have more than you can hold in your hand, regardless of whether the box will hold them or not, you'll have to get rid of some of the cards. This is exactly the problem we're faced with when using a single recorder and console basic. We can open a file in only one of two modes - INPUT or OUTPUT. Once we have INPUTted previous data and stored it in an array, we <u>can</u> add more items to the array, provided the additional information doesn't cause us to run out of memory. We can also take items out of the array or change values within the array. After all of the information is changed, added to, or rearranged to our liking, <u>then</u> we can OUTPUT the corrected and updated information back to the data file.

<u>Summary.</u> Following is a summary of some of the more important points mentioned in the above discussion. All of these apply only to those operating with a single recorder.

1. For efficiency, test all data, structure it to specific lengths, and change all data to string data prior to building your record.

2. Join all individual string data together, i.e. RECORD$=A$&B$&C$, and print the entire record to the data file as one long string.

3. Try to make utilization of the longest record length possible (192).

4. Input complete records and use "loops" to subdivide the record and turn it back into individual variables.

5. To add, change, or update information, all previous information must be brought into memory first.

Except for item 5 above, there are other options regarding how to transfer information to and from data files. Following is some justification for recommending this method, as well as some information on what future expansions might do for your capabilities.

<u>Justification & Explanation.</u> Much has been said and numerous examples are given in your user's manual about how to format your PRINT and INPUT statement using commas as separators, and the advantages and disadvantages of INTERNAL over DISPLAY. There may be some particular applications where their method of printing data might be superior to the method we described above; however, as a universal tool, we feel the practice of printing a single string, and inputting a single string, is far superior. When dealing with a cassette recorder and console basic, we essentially have 2 factors to consider in order to develop the best possible data filing system. These factors are time and memory.

<u>Time.</u> It should be obvious that the most time consuming factor working with data files is the time required to print to and read from the cassette recorder. The following example will demonstrate why we recommend using maximum length records for this purpose:

```
>100 L=190
>110 IF LEN(X$)=L THEN 140
>120 X$=X$&"X"
>130 GOTO 110
>140 PRINT X$
>150 OPEN #1:"CS1",INTERNAL,O
 UTPUT,FIXED 192
>160 FOR I=1 TO 10
>170 PRINT #1:X$
>180 NEXT I
>190 CLOSE #1
```

This program builds a data line 190 characters long, displays it to the screen, opens a file, and prints the 190 characters to the cassette ten times. Total run time, from the first time you hear it start to print, until it stops is approximately 1 Min 25 Sec. If you divide the 85 seconds by 1,900 characters this works out to approximately .0447 seconds per character. If you change line 100 to L=60 and remove the 192 from Line 150, the run time decreases to about 1 Min 5 Sec. The same calculation (65 seconds/600 characters) results in a time of .1083 per character. The first method transfers "raw" characters approximately 2.4 times as fast as the second method. On the input side you will find the calculations almost identical. It should be obvious, for sheer movement of data, the use of a full 192 character record is the most efficient method. To further increase the speed, you should always use the INTERNAL mode and not DISPLAY; however, the difference is really very slight. By changing that command in the above program, unless you have a very good stopwatch, the difference is easier to "hear" than it is to measure in terms of seconds.

If you agree that moving characters quickly is an advantage, then you should also agree that the next trick would be to pack as much information (or elements of data) on a 192 character record as possible. Here again, our method proves superior. Look at the following example:

```
>100 OPEN #1:"CS1",INTERNAL,O
 UTPUT,FIXED
>110 A=20
>120 B=300
>130 C=1
>140 D$="TEST"
>150 PRINT #1:A,B,C,D$
>160 CLOSE #1
>170 OPEN #1:"CS1",INTERNAL,I
 NPUT ,FIXED
>180 INPUT #1:A,B,C,D$
>190 PRINT A:B:C:D$
>200 CLOSE #1
```

If we opened the above file in the INTERNAL, OUTPUT mode, and we printed the above information to the file it would require 32 characters out of the basic 64 character line. In this mode, each "number" takes up 8 spaces, plus one for "overhead", and each string takes up its exact length, plus one for "overhead". Adding it all up comes to 32 spaces used. You can prove this is true by adding more numeric variables until the program errors out because the default line of 64 will not hold it. It would hold only 4 more numeric variables and then start overflowing to the next data record. On the input side you would get an error message.

If you used the DISPLAY, OUTPUT mode, this information would use up 15 spaces. You can verify this by changing the INTERNAL to DISPLAY and then changing 180 to INPUT #1:X$. Change 190 to read PRINT X$;LEN(X$) and you'll see the data line and length.

Using the method we describe, each variable is changed to a string and the result is just one string, only ten characters long ("203001TEST"). If you add one more space for overhead, it still uses only 11 spaces. In the bowling program, where we've added 12 or 13 groupings of information together, we still only have 1 byte of overhead. The fact is, more items can be "packed" on a 192 character line using our method than any other method.

Memory. Remembering that you can't use a cassette recorder in the UPDATE or APPEND mode, what you can do with data files and one recorder ultimately boils down to how much information you can hold in memory. Let's use an example to emphasize this point.

If you recall, in Chapter 2 we said that any program involving data files "begins with deciding what information is needed and how it will be stored". If we wanted to build a mailing list, we might decide to set up a data file that allowed: 25 spaces for name, 25 for address, 15 for City, 2 for State, 5 for ZIP code, 12 for Telephone, and 10 for Miscellaneous info. This means that each entry is going to consume 94 characters. Double that, and we arrive at 188, or about the length of one data record. Suppose we had 150 names and we wanted to be able to sort and display them in ZIP code order or alphabetical order. We also want to have the ability to change, add, and/or delete items. Can it be done with one recorder? A simple calculation of 150 (names) X 94 (char/entry) shows that this would require approx 14,100 bytes of memory. Since we start with 16K and the CPU eats about 2,000 immediately, we only have about 14,000 bytes available when we first turn on the system. Whatever program you're running must share the remaining 14,000 bytes with the data that it's going to have to hold in memory, and even a fairly simple program is going to consume 4,000-5,000 bytes. If we can't get it all in memory, the only solution is to break our file down into several smaller files which can be accessed individually. We might use: one cassette for A-F; another for G-N; and still another for O-Z. When you want to add/change/delete you simply: load the appropriate cassette; INPUT all information into memory; do your update; sort it again; and OUTPUT it back to the cassette. This works fine for the alpha report, but what about sorting it in ZIP code order?

If you designed the list, you probably have some idea of the range of ZIP codes, i.e. they might run from 40301 to 50201. When you wanted a ZIP sort, your program would have to ask you, "What Range? ". You would have to give it two numbers, close enough together that the total number of names within that range doesn't exceed about 50. With this information, the program should then ask you to, "Load A-F file. The computer would check each record sequentially to see if it was within range and bring it into memory only if it was. That information would be stored in an array. When it reaches the end of that cassette the program should then ask if you have any more cassettes. If you answer "Yes", it will ask you

to load it, and this process would continue until all cassettes are searched. After the last cassette is read, the computer should have an array in memory containing all ZIP codes within your specified range. After running it through a SORT routine, the names and address for that group of ZIP codes can be displayed in order.

The above example shows how it can be done. Whether it's practical, or whether you're willing to go to the bother, is for you to decide. Searching a single cassette of 5Ø names (25 records with 2 on each record) will require about 3.5 minutes. With 3 cassettes that's 1Ø.5 minutes of read time, plus your handling time between cassettes.

Before we leave this subject of memory, let's discuss briefly how to store the information in the array. Numeric arrays, just like the numeric information printed to the data files, automatically use up 8 bytes of memory for each element as soon as they are dimensioned. For instance, when bringing in the ZIP codes, you could dimension an array called ZIP(5Ø). This automatically eats up a minimum of 4ØØ bytes, plus overhead, whether you ever put a value in the array or not. If you use a string array such as ZIP$(5Ø), no memory is consumed until you start filling the array. When you start filling it, each element will only use only 5 bytes of memory (the length of the string). As you'll see in the chapter on sorting, string data can be sorted as easily as numeric, so these never need to be converted back to numeric variables. We're not going to go into any greater detail on sorting and arrays in this chapter; however, they are covered fully in chapters 7 and 8.

Headers. Often times it's convenient to use the first two or three records (lines of data) of a specific data file to hold certain information which: may be required frequently; which speeds the operation of the program; or which is only used once.

The data file created for storing your personal checkbook information has header data in the first 4 lines. On those four lines we store: the names of all income and expense accounts; the budget figure for each account; the year-to-date total for each account; the bank balance; and the last entry number. That program builds a perpetual file of all of your personal checks with up to 71 on a cassette (the same idea as subdividing your mailing list). Without this "header" data, each time we wanted a total for an account we would have to load each cassette, and read through every entry, in order to arrive at the year-to-date figure for that account. The same is true of Bank Balance. The Budget Display program doesn't need any detailed data on individual checks. In order to display your year-to-date and budget graph all it needs to do is INPUT four records from your latest data cassette.

Recalling the loops that we set up to analyze these records, how is the program supposed to know how many records it has to analyze? In the bowling program we know that we always have 18 lines of data; but in a mailing list program this figure may be constantly changing. By storing some information in the first record, you can use that data to adjust your FOR - NEXT statment. When you start your mailing list you would have a value of zero for number of entries (call it ENTRY or some other variable name). As each new name is added or

deleted you adjust this value. When you have updated a file, you count the entries in your array and determine how many records are required. PRINT that value to your data file first and then use the variable in your PRINT loop, i.e. FOR I=1 TO ENTRY. The same thing works in reverse when reading in data.

End of Entry or File. The way our bowling program is constructed, it always OUTPUTs 18 entries and always INPUTs 18 entries. This is not necessarily required. As part of your OUTPUT program, you could have the program print a value of something like "XX" in a specific spot following its last entry. On the INPUT side, you set your loop for the maximum number of records, but have provisions for it to terminate when it finds the "XX". The Checkbook program uses this technique. At any point in time, the data file is only long enough to hold 4 header records, plus the number of records required to hold the total number of checks in that file. If you had only 1Ø checks active it wouldn't take as long to load the information as if you had 6Ø checks active. In order to provide and check for the "XX" or other end of file designation you'll have to code in more program lines. The more program lines you use, the less room you'll have to store data in memory.

Expanding Your System

You may have realized from the above discussion that some applications are just too big for a single recorder. The next thing you need to know is how you should expand your system for greater capability. We're not going to go into a detailed explanation of these expansions, but we feel you should know what they can do for you.

Second Recorder. Adding a second recorder will not increase the speed with which you can read or print data: all of the same time limitations still exist. However, you can build longer complete files. All of the 15Ø name mailing list described above could be kept on one cassette. Let's say you wanted to add 2 names, delete one, and change one. Your program could accept this information from the keyboard and hold it in memory. It would then open cassette 1 (CS1) in the INPUT mode and cassette 2 (CS2) in the OUTPUT mode. After each input from CS1 the program would go through a series of IF statements. If the record was one that needed to be deleted, it would go on to the next record and it would not print it to the new file being created on CS2. If it was one that needed to be changed, the information would be adjusted accordingly, and then it would be printed to the new file on CS2. If your file was alphabetical, it would also be looking for the appropriate time to "drop in" the new records so that the final file on CS2 is in order.

Expanded Memory. Because we have to bring all information into memory in order to sort and handle data files with a single cassette, the advantage of additional memory is readily apparent. If you can live with the time element of inputting 75 or 8Ø records from a cassette recorder, with expanded memory it's possible to hold, sort, and manipulate perhaps 2ØØ or more records like those described for the mailing list. On a personal checkbook program we could probably keep 25Ø or 3ØØ of your last checks in memory.

Disk Drive. A powerful tool! Because of its speed, the availability to open in the UPDATE and APPEND mode, and the ability to use RELATIVE files with the RECord command, this is the only answer for situations requiring large data files. Remember our 3 X 5 box? With disk drive you can do all the things that we couldn't do before, and you can do it hundreds of times faster. We can add a card to the back of the file; put one in the middle and move all the rest back; or take one out and throw it away. Because of its speed, even additional memory is not as important any more. To sort a file containing names and addresses you would simply "rip" through the file pulling out the ZIP code for each name and store that in memory along with just the record number of that entry. You might wind up with an array called ZIP(X) where the values of ZIP(1 through 4) are: 3030215, 3050703, 4020973, 5080912. The first five digits of each value can represent the ZIP and the last two digits the record number. Now all you have to do is: read down your array (FOR I=1 TO 4); get your record number, get that record from the disk; and print it or process it as desired.

Things To Work On. Learning to work with data files requires a clear understanding of: how to convert numbers to string data, STR$(n); string data to numbers, VAL(X$); FOR - NEXT - STEP; and numeric and string arrays. There are numerous examples of all three in the programs provided thus far, but remember that each file is unique and the loops required for each will vary considerably. The method we use is fairly easy to debug since you can put a PRINT statement directly in front of the PRINT #1 statement and see what's being sent to file -- exactly as it looks in the file. The same holds true on the input side. Since you decide in advance where everything goes, it's normally pretty easy to spot a mistake and correct it.

```
*************************
*   BUDGET MAINTENANCE  *
*        V-PC232KB      *
*      BY  T CASTLE     *
*************************

*************************
*   BUDGET/YTD DISPLAY  *
*        V-PD231KB      *
*      BY  T CASTLE     *
*************************
```

DESCRIPTION - Budget Maintenance. The "Budget/YTD Display", work hand-in-hand to provide the user with the foundation for a home financial management system. The first of the two programs, "Budget Maintenance", provides for the creation of a "Chart of Accounts" which includes: the name of each account; a monthly budget figure for each of the expense/income accounts; and a cumulative total showing the Year-to-Date amount charged to each account. It also makes provisions for the entering, retrieval, and updating, of up to 71 individual checkbook entries (either checks, deposits, or adjustments). When a total of 71 entries have been made, or at some other convenient breaking point, the cumulative budget and YTD figures, as well as the data on individual entries, can be stored on cassette tape for later retrieval and processing by other programs. Following is a more detailed discussion of the "Budget Maintenance" Program.

DETAILED DISCUSSION. This program was designed to be functional, rather than pretty, colorful, or exciting. Just about all REMarks and other unnecessary lines and letters have been removed from the program to permit a maximum amount of information to be maintained "in memory". The result of this trimming is that the variable names are not necessarily descriptive. It would have been nice to have the bank balance named BKBAL; however, BB had to be sufficient since it occurs frequently and would require 3 more bytes of memory for each occurrence. The main variables in use are the array variables dimensioned in lines 130 and 140. There are a total of 35 possible budget records, each one containing: a name (A$); a monthly budget amount (B$); and a YTD cumulative figure (C$). There are a total of 71 possible check/deposit records, each one containing: a check or reference number (D$); the amount of the check/deposit (E$); the date of the entry (F$); the name of the person to whom the check was written or from whom it was received (G$); and the budget account which it affects (H$).

Almost all input is received initially as a string variable named Q$. A four digit code is then added to each input and the resulting Q$ is then sent through a validation subroutine in lines 3460-4060. The first digit of the code represents the type of entry that it should be. "A" indicates that it should be a budget account between 1 and 35; "R" indicates that it should be a check/deposit record number between 1 and 71; "C" indicates that it should be a dollars and cents type number with two places following the decimal; "D" indicates that it should be a date; and "N" that it should be a whole number (integer), with no decimals. The second two digits tell the subroutine how long the string should be in order to be consistent with all others in the array. The third digit indicates whether it should be right or left justified. On RETURN from the subroutine Q$ is either valid and properly formatted or it is returned with a value of "X". If "X" is returned, the answer is

rejected and the user must reenter a new response.

The main controlling section of the program is found in lines 610-910 and it consists of the "Main Menu" with six options. The first option is for inputting new check/deposit records and operates through the subroutines in lines 920-1170 and 2090-2810. It keeps track of what the next available record number is, it permits sequential entering of each element of the record, and makes provisions for verification of each element by the user prior to updating the bank balance and year-to-date figures. The second option is for scanning and/or changing check/deposit records and generally operates through the subroutines in lines 1750-2080 and 2090-2810. It can display all 71 records (or as many as are active in the current file) in groups of 5. After a group of 5 is displayed the user is given the option of selecting one for changing and eventually the user can change any element of the entry. After RETURN to the main menu, the affected budget figure and/or bank balance is updated accordingly. The third option is for scanning and/or changing Budget/YTD information and generally operates through the subroutine located in lines 1180-1740. It can display all 35 budget accounts, including the YTD figure and monthly budget estimate. It provides for scanning in groups of five. Until the user establishes account names, the accounts are pre-named (1-5) Income and (6-35) Expense. Zero balances are established for all monthly budget amounts and year-to-date balances. At the end of an update or posting session, option four is used to SAVE the information, in its current state, on data cassette. It operates through the subroutine located in lines 3170-3450. It gathers the first 9 elements of the A$, B$, and C$ arrays and combines them to form one fixed length data line 192 long. This is then printed to the data cassette. A second and third line is then constructed, each containing 9 elements, and these too are printed to the file. Finally a line is created consisting of 8 Budget elements, followed by the Start Date, Bank Balance, and Last Record Number. This line is then printed to the file. Following this, the subroutine gathers check/deposit records in groups of 6 and prints them to the file until it reaches the first unused record, at which time the subroutine ends and no further unused records are printed to the file. Option five permits the user to clear or empty all of the check/deposit data while retaining the current status of all budget and YTD information. It's logical to store data on casssette in convenient groups. For instance, if you have approximately 30 check/deposit transactions per month, you may put about two months worth of records on a data cassette. If you begin a new month and the first available record number is number 60, you know that you can't get another whole months' worth of records on that cassette. Utilize option 4 to save your records through the end of the previous month and begin a new month after using option five to clear the entries. Option six simply instructs the user to do a "Function/Quit" to exit the program.

<u>DESCRIPTION - Budget/YTD Display.</u> The Budget/YTD Display program operates on the data created and stored with the Budget Maintenance Program. The first thing the program asks the user to do is to load the latest data cassette containing the Budget and Year-to-Date information. The program loads the

first 4 lines of this file into memory (the lines containing the budget/YTD information). As it's loading the data for each account, it checks to see if there is a value, other than zero, in either the Budget or YTD category. If it finds an amount it saves the account number in an array named "A".

After all accounts are loaded the program displays the program and then asks for the "AS OF DATE". The program requires this information in order to calculate an average cost per month utilizing the start date and year-to-date totals. After this is done, the program clears the screen and displays the account numbers and account names for all "active" accounts. The user is instructed to pick up to five accounts to be displayed on a bar graph. If you want to see less than five you can enter zeros in place of numbers. The screen then clears and is replaced by a white graph with grey lines. The grid is outlined in black and rests on a yellow background. At the top and to the right there is the word "BUDGET" preceded by a GREEN block and the word "AVERAGE" preceded by a RED Block. On the next line and every fourth line thereafter, the name of the account selected, the Budget Amount, and the Average Amount spent per month is displayed. On the two lines immediately below each account a bar graph is created showing the BUDGET figure in Green and the AVERAGE figure in Red. In order to view other categories the user simply needs to hit any key and the program cycles back to the list of active accounts. The lines created are always in relation to each other. For any group of five selected, the computer determines the longest line and sets the increment per square based on this figure. The length of each line is then calculated to the nearest 1/4 of a block and the appropriate length line is created. For this reason, the graph appears more realistic and is more accurate if the five items selected all have budget and/or YTD figures in approximately the same range. Graphing an income account such as your pay check against a yard maintenance account would wind up with an extremenly short line for yard maintenance.

<u>Notes.</u> The first thing that must be done if you are planning on creating a financial management system is that the programs must be created and "debugged". Once you have entered the programs and they "appear" to be operating properly, test and re-test them until you are confident that each and every year-to-date figure and the bank balance are being properly updated. Enter test information using the number one option and then check the status of the budget accounts using the number 3 option; try entering negative numbers; switch account numbers using the number 2 option; change YTD figures using the number 3 option. In general, before you begin to enter actual data, make it a point to understand exactly what it is that is supposed to take place in the program.

Once you are satisifed with the accuracy of the program, the next thing to do is establish a "Chart of Accounts". We have provided a suggested chart following this section to give you some idea what it might look like. Try to select accounts for which you usually write a single check. If you have two cars, and try to create a gas account for each auto, you may have to make a lot of manual adjustments to the year-to-date data

since you might pay your credit card bill with a single check. It makes more sense to allow "Food" to cover everything puchased at the grocery store, as opposed to trying to keep a separate figure for food, cigarettes, sundries, etc. The information obtained from a system like this is only as good as the information you give it. If you do have some checks which you know are going to be distributed to a number of sources, such as department store credit cards, always code them to a holding account such as "Crd Card". At then end of each posting session, make it a point to clear this account and redistribute it amoung the affected expense accounts.

Plan Your Posting. Be consistent! Set aside a certain day each week when you update your records -- sit down and code them all in at one time. Do it regularly! If you get behind you may find it quite a chore to catch up. Always keep a handwritten or typed record of each entry made. Unless your computer is tied in to a printer and you have the ability to obtain a hard copy printout of your entries, do not rely on the information in the data file as your only source. Always save your data twice before closing your program and keep each copy on a separate cassette. Carefully label these and store them in different places.

WARNING! While every effort is made to insure the accuracy of these programs, AMLIST, Inc., can assume no liability for losses or damages which may occur as a result of reliance on the information provided herein.

CHART OF ACCOUNTS

Income.

Ø1 -	Pay #1
Ø2 -	Pay #2
Ø3 -	Extra Inc
Ø4-Ø5	Not Used

Expenses.

Ø6 -	Food
Ø7 -	Clothing
Ø8 -	Utilities
Ø9 -	Telephone
1Ø -	Auto Gas&Repair
11 -	Auto Replacment
12 -	Mortgage
13 -	Furnishings
14 -	Improvements
15 -	Misc Household
16 -	Misc Yard
17 -	Entertainment
18 -	Charity
19 -	Courtesy
2Ø -	Savings
21 -	Insurance
22 -	Dental
23 -	Medical
24 -	Child Care
25 -	Travel
26-29	Not Used
3Ø -	Credit Cards
31-34	Not Used
35-	Miscellaneous

```
100 REM  BUDGET MAINTENANCE
110 REM  BY T CASTLE
120 REM  AMLIST V-PC232KB
130 DIM A$(35),B$(35),C$(35)
,D$(72)
140 DIM E$(72),F$(72),G$(72)
,H$(72)
150 CALL CLEAR
160 PRINT "DO YOU HAVE AN EX
ISTING DATA";"CASSETTE TO WO
RK FROM"::::::::
170 INPUT "Y (YES) OR N (NO)
: ":Q$
180 IF Q$="Y" THEN 200
190 IF Q$="N" THEN 290 ELSE
170
200 CALL CLEAR
210 PRINT "LOAD LATEST BUDGE
T CASSETTE":::"HIT ANY KEY":
:
220 CALL KEY(3,KY,ST)
230 IF ST=0 THEN 220
240 OPEN #1:"CS1",INTERNAL,I
NPUT ,FIXED 192
250 GOSUB 2820
260 GOSUB 3000
270 CLOSE #1
280 GOTO 610
290 CALL CLEAR
300 INPUT "START DATE(010183
): ":Q$
310 Q$="D06L"&Q$
320 GOSUB 3460
330 IF Q$="X" THEN 300
340 SR$=Q$
350 INPUT "STARTING BALANCE:
 ":Q$
360 Q$="C10R"&Q$
370 GOSUB 3460
380 IF Q$="X" THEN 350
390 BB$=Q$
400 LT$="00"
410 FOR I=1 TO 35
420 IF I<6 THEN 450
430 A$(I)="EXPENSE "
440 GOTO 460
450 A$(I)="INCOME  "
460 B$(I)="   00"
470 C$(I)="    0.00"
480 NEXT I
490 NX=1
500 GOSUB 520
510 GOTO 610
520 NX=VAL(LT$)+1
530 D$(NX)="0000"
540 E$(NX)="0000.00"
550 F$(NX)="00000"
560 G$(NX)="XXXXXXXXXX"
570 H$(NX)="00"
580 IF NX=72 THEN 590 ELSE 6
00
590 STP=1
600 RETURN
610 CALL CLEAR
620 PRINT TAB(9);"MAIN MENU"
::::"BALANCE= ";BB$::"1 -NEW
 CHECK/DEP":"2 -CHANGE CHECK
/DEP"
630 PRINT "3 -CHANGE BDGT/YT
D":"4 -SAVE DATA":"5 -CLEAR
CHECK/DEP":"6 -EXIT PROGRAM"
:::::::
640 INPUT "SELECTION? ":Q$
650 CALL CLEAR
660 Q$="N01L"&Q$
670 GOSUB 3460
680 IF Q$="X" THEN 640
690 MQ=VAL(Q$)
700 ON MQ GOTO 710,730,750,7
70,790,900
710 GOSUB 920
720 GOTO 610
730 GOSUB 1750
740 GOTO 610
750 GOSUB 1180
760 GOTO 610
770 GOSUB 3170
780 GOTO 610
790 PRINT "CLEARING ENTRIES"
800 LT$="0"
810 GOSUB 520
820 FOR I=2 TO 72
830 D$(I)=""
840 E$(I)=""
850 F$(I)=""
860 G$(I)=""
870 H$(I)=""
880 NEXT I
```

```
890 GOTO 610
900 PRINT "FUNCTION/QUIT TO
EXIT"
910 GOTO 910
920 CALL CLEAR
930 IF STP=1 THEN 1170
940 C=VAL(LT$)+1
950 W$="1"
960 GOSUB 2090
970 W$="0"
980 CX=0
990 TC=VAL(H$(C))
1000 TT=VAL(E$(C))
1010 IF TC>5 THEN 1040
1020 BB=VAL(BB$)+TT
1030 GOTO 1050
1040 BB=VAL(BB$)-TT
1050 Q$="C10R"&STR$(BB)
1060 GOSUB 3460
1070 BB$=Q$
1080 YTD=VAL(C$(TC))+TT
1090 Q$=STR$(YTD)
1100 Q$="C08R"&Q$
1110 GOSUB 3460
1120 C$(TC)=Q$
1130 GOSUB 2090
1140 LT$=STR$(C)
1150 NX=VAL(LT$)
1160 GOSUB 520
1170 RETURN
1180 CALL CLEAR
1190 PRINT "ACCTS ARE NUMBER
ED  1 - 35":::::"THEY DISPLA
Y IN GROUPS OF 5":::::::
1200 INPUT "NO. 1-31 OR R FO
R MENU: ":Q$
1210 IF Q$="R" THEN 1740
1220 Q$="A02R"&Q$
1230 GOSUB 3460
1240 IF Q$="X" THEN 1200
1250 C1=VAL(Q$)
1260 IF C1>31 THEN 1270 ELSE
 1280
1270 C1=31
1280 CALL CLEAR
1290 PRINT "#  NAME      BDG
T      YTD"::
1300 FOR I=C1 TO C1+4
1310 IF I>9 THEN 1330
1320 PRINT " ";
1330 PRINT STR$(I);" ";A$(I)
&" ";
1340 PRINT B$(I);" ";C$(I)
1350 NEXT I
1360 PRINT :::
1370 INPUT "# TO CHANGE / R
TO MENU: ":Q$
1380 IF Q$="R" THEN 1180
1390 Q$="A02R"&Q$
1400 GOSUB 3460
1410 IF Q$="X" THEN 1370
1420 C=VAL(Q$)
1430 CALL CLEAR
1440 PRINT "#  NAME      BDG
T      YTD"::
1450 IF C>9 THEN 1470
1460 PRINT " ";
1470 PRINT STR$(C);" ";A$(C)
;" ";
1480 PRINT B$(C);" ";C$(C)
1490 PRINT ::"N -CHANGE NAME
":"B -CHANGE BUDGET":"Y -CHA
NGE YTD":"R - MENU"::

1500 INPUT "ANSWER? ":Q4$
1510 PRINT
1520 IF Q4$="N" THEN 1560
1530 IF Q4$="B" THEN 1620
1540 IF Q4$="Y" THEN 1680
1550 IF Q4$="R" THEN 1280 EL
SE 1500
1560 INPUT "NAME?   ":Q$
1570 Q$="S08L"&Q$
1580 GOSUB 3460
1590 IF Q$="X" THEN 1560
1600 A$(C)=Q$
1610 GOTO 1430
1620 INPUT "BUDGET? ":Q$
1630 Q$="N05R"&Q$
1640 GOSUB 3460
1650 IF Q$="X" THEN 1620
1660 B$(C)=Q$
1670 GOTO 1430
1680 INPUT "YTD?    ":Q$
1690 Q$="C08R"&Q$
1700 GOSUB 3460
1710 IF Q$="X" THEN 1680
1720 C$(C)=Q$
1730 GOTO 1430
1740 RETURN
1750 CALL CLEAR
```

```
1760 PRINT "MAXIMUM OF 71 CH
ECK RECORDS"::::"THEY DISPLA
Y IN GROUPS OF 5"::::::::
1770 INPUT "NO. 1-67 OR R FO
R MENU: ":Q$
1780 IF Q$="R" THEN 2080
1790 Q$="R02R"&Q$
1800 GOSUB 3460
1810 IF Q$="X" THEN 1770
1820 C1=VAL(Q$)
1830 IF C1>67 THEN 1840 ELSE
 1850
1840 C1=67
1850 CALL CLEAR
1860 PRINT " # TO OR SOURCE"
:"   DATE   REC#   AMOUNT   ACC
T"::
1870 FOR I=C1 TO C1+4
1880 IF F$(I)="" THEN 1890 E
LSE 1910
1890 DT$=""
1900 GOTO 1920
1910 DT$=SEG$(F$(I),2,4)&"8"
&SEG$(F$(I),1,1)
1920 PRINT STR$(I)&" "&G$(I)
:"   "&DT$&" ";
1930 PRINT D$(I)&" "&E$(I)&"
   "&H$(I)
1940 NEXT I
1950 INPUT "# TO CHANGE / R
TO MENU: ":Q$
1960 IF Q$="R" THEN 1750
1970 Q$="R02R"&Q$
1980 GOSUB 3460
1990 IF Q$="X" THEN 1950
2000 IF VAL(Q$)>VAL(LT$)THEN
 1950
2010 C=VAL(Q$)
2020 TC=VAL(H$(C))
2030 TT=VAL(E$(C))
2040 YTD=VAL(C$(TC))
2050 BB=VAL(BB$)
2060 GOSUB 2100
2070 GOTO 1850
2080 RETURN
2090 IF CX>0 THEN 2260
2100 CALL CLEAR
2110 PRINT "BANK BALANCE: ";
BB$::" # TO OR SOURCE":"
DATE  REC#  AMOUNT  ACCT"::
2120 DT$=SEG$(F$(C),2,4)&"8"
&SEG$(F$(C),1,1)
2130 PRINT STR$(C)&" "&G$(C)
2140 PRINT "    "&DT$&" ";D$(
C)&" ";
2150 PRINT E$(C)&"   "&H$(C)
::::
2160 IF W$="1" THEN 2260
2170 PRINT "ENTER # TO CHANG
E"::"1-TO/SOURCE   2-DATE":"
3-CK/DEPOSIT  4-AMOUNT":"5-A
CCOUNT      6-MENU"::
2180 INPUT "ANSWER? ":Q$
2190 PRINT
2200 Q$="N01L"&Q$
2210 GOSUB 3460
2220 IF Q$="X" THEN 2180
2230 Q4=VAL(Q$)
2240 IF (Q4<1)+(Q4>6)THEN 21
80
2250 ON Q4 GOTO 2280,2340,24
20,2480,2540,2600
2260 CX=CX+1
2270 ON CX GOTO 2280,2340,24
20,2480,2540,2810
2280 INPUT "NAME?  ":Q$
2290 Q$="S10L"&Q$
2300 GOSUB 3460
2310 IF Q$="X" THEN 2280
2320 G$(C)=Q$
2330 GOTO 2090
2340 INPUT "DATE?  ":Q$
2350 Q$="D06L"&Q$
2360 GOSUB 3460
2370 IF Q$="X" THEN 2340
2380 DT$=SEG$(Q$,6,1)
2390 DT$=DT$&SEG$(Q$,1,4)
2400 F$(C)=DT$
2410 GOTO 2090
2420 INPUT "NUMBER?":Q$
2430 Q$="S04L"&Q$
2440 GOSUB 3460
2450 IF Q$="X" THEN 2420
2460 D$(C)=Q$
2470 GOTO 2090
```

```
2480 INPUT "AMOUNT?":Q$
2490 Q$="C07R"&Q$
2500 GOSUB 3460
2510 IF Q$="X" THEN 2480
2520 E$(C)=Q$
2530 GOTO 2090
2540 INPUT "ACCOUNT?":Q$
2550 Q$="A02L"&Q$
2560 GOSUB 3460
2570 IF Q$="X" THEN 2540
2580 H$(C)=Q$
2590 GOTO 2090
2600 NC=VAL(H$(C))
2610 NT=VAL(E$(C))
2620 DF=TT-NT
2630 IF DF=0 THEN 2710
2640 IF NC<5 THEN 2670
2650 NB=BB+DF
2660 GOTO 2680
2670 NB=BB-DF
2680 Q$="C10R"&STR$(NB)
2690 GOSUB 3460
2700 BB$=Q$
2710 IF NC=TC THEN 2810
2720 HTD=YTD-TT
2730 Q$="C08R"&STR$(HTD)
2740 GOSUB 3460
2750 C$(TC)=Q$
2760 NYD=VAL(C$(NC))
2770 HTD=NYD+NT
2780 Q$="C08R"&STR$(HTD)
2790 GOSUB 3460
2800 C$(NC)=Q$
2810 RETURN
2820 REM
2830 FOR I=1 TO 28 STEP 9
2840 C=-20
2850 INPUT #1:X$
2860 K=I+8
2870 IF K>35 THEN 2880 ELSE
2890
2880 K=K-1
2890 FOR J=I TO K
2900 C=C+21
2910 A$(J)=SEG$(X$,C,8)
2920 B$(J)=SEG$(X$,C+8,5)
2930 C$(J)=SEG$(X$,C+13,8)
2940 NEXT J
2950 NEXT I
2960 SR$=SEG$(X$,169,6)
2970 BB$=SEG$(X$,175,10)
2980 LT$=SEG$(X$,185,2)
2990 RETURN
3000 FOR I=1 TO 67 STEP 6
3010 C=-27
3020 INPUT #1:X$
3030 K=I+5
3040 FOR J=I TO K
3050 C=C+28
3060 D$(J)=SEG$(X$,C,4)
3070 E$(J)=SEG$(X$,C+4,7)
3080 F$(J)=SEG$(X$,C+11,5)
3090 G$(J)=SEG$(X$,C+16,10)
3100 H$(J)=SEG$(X$,C+26,2)
3110 IF G$(J)="XXXXXXXXXX" T
HEN 3120 ELSE 3140
3120 J=K
3130 I=67
3140 NEXT J
3150 NEXT I
3160 RETURN
3170 PRINT "INSERT NEW DATA
CASSETTE"::"HIT ANY KEY":::
3180 X$=""
3190 CALL KEY(3,KY,ST)
3200 IF ST=0 THEN 3190
3210 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
3220 FOR I=1 TO 28 STEP 9
3230 K=I+8
3240 IF K>35 THEN 3250 ELSE
3260
3250 K=K-1
3260 FOR J=I TO K
3270 X$=X$&A$(J)&B$(J)&C$(J)
3280 NEXT J
3290 IF I=28 THEN 3300 ELSE
3310
3300 X$=SEG$(X$,1,168)&SR$&B
B$&LT$
3310 PRINT #1:X$
3320 X$=""
3330 NEXT I
3340 FOR I=1 TO 67 STEP 6
3350 K=I+5
3360 FOR J=I TO K
3370 X$=X$&D$(J)&E$(J)&F$(J)
&G$(J)&H$(J)
3380 IF G$(J)="XXXXXXXXXX" T
HEN 3390 ELSE 3400
```

```
3390 I=67
3400 NEXT J
3410 PRINT #1:X$
3420 X$=""
3430 NEXT I
3440 CLOSE #1
3450 RETURN
3460 VN=0
3470 VD=0
3480 VP=0
3490 NL=VAL(SEG$(Q$,2,2))
3500 L=LEN(Q$)-4
3510 VT$=SEG$(Q$,1,1)
3520 VJ$=SEG$(Q$,4,1)
3530 IF L=0 THEN 4030
3540 R$=SEG$(Q$,5,L)
3550 IF VT$="S" THEN 3950
3560 FOR VC=1 TO L
3570 VL=ASC(SEG$(R$,VC,1))
3580 IF (VL<48)+(VL>57)THEN
3600
3590 GOTO 3650
3600 IF (VL<45)+(VL>46)THEN
3640
3610 IF VL=45 THEN 3650
3620 VD=VD+1
3630 VP=VC
3640 VN=VN+1
3650 NEXT VC
3660 IF VT$="C" THEN 3850
3670 IF VN>0 THEN 4030
3680 V1=VAL(R$)
3690 IF VT$="A" THEN 3730
3700 IF VT$="R" THEN 3750
3710 IF VT$="D" THEN 3770
3720 GOTO 3950
3730 IF (V1<1)+(V1>35)THEN 4
030
3740 GOTO 3950
3750 IF (V1<1)+(V1>71)THEN 4
030
3760 GOTO 3950
3770 IF L<>NL THEN 4030
3780 V2=VAL(SEG$(R$,1,2))
3790 V3=VAL(SEG$(R$,3,2))
3800 V4=VAL(SEG$(R$,5,2))
3810 IF (V2<1)+(V2>12)THEN 4
030
3820 IF (V3<1)+(V3>31)THEN 4
030
3830 IF (V4<82)+(V4>90)THEN
4030
3840 GOTO 4050
3850 IF VN>1 THEN 4030
3860 IF VP>0 THEN 3900
3870 R$=R$&".00"
3880 L=L+3
3890 GOTO 3950
3900 IF L-VP=2 THEN 3950
3910 IF L-VP>2 THEN 4030
3920 IF L-VP=1 THEN 3930 ELS
E 4030
3930 R$=R$&"0"
3940 L=L+1
3950 IF L>NL THEN 4030
3960 IF L=NL THEN 4050
3970 IF VJ$="R" THEN 4000
3980 R$=R$&" "
3990 GOTO 4010
4000 R$=" "&R$
4010 L=LEN(R$)
4020 GOTO 3960
4030 Q$="X"
4040 GOTO 4060
4050 Q$=R$
4060 RETURN
```

HAPPY COMPUTING!

```
100 REM  ******************
110 REM  BUDGET/YTD DISPLAY
120 REM  ******************
130 REM  AMLIST V-PD231KB
140 REM  BY T CASTLE
150 REM  SET VALUES
160 GOSUB 370
170 REM  OPEN FILE/GET DATA
180 GOSUB 610
190 REM  DISPLAY ACCOUNTS
200 GOSUB 910
210 REM  SCREEN DISPLAY 1
220 GOSUB 1090
230 REM  CALC AVERAGES
240 GOSUB 1830
250 REM  ADD DATA TO DISPLAY
260 GOSUB 1240
270 REM  CALCULATE INCREMENT
280 GOSUB 2010
290 REM  PRINTS BAR GRAPH
300 GOSUB 2170
310 MSG$="2406HIT[ANY[KEY[TO
[RETURN"
320 GOSUB 2560
330 CALL KEY(3,KY,ST)
340 IF ST=0 THEN 330
350 GOTO 190
360 REM  SET INITIAL VALUES
370 CALL CLEAR
380 DIM AP$(35),BP$(35),CP$(
35)
390 DIM A(35),A$(35),B$(35),
C$(35)
400 CALL CHAR(38,"FFFFFFFFFF
FFFFFF")
410 CALL CHAR(128,"FFC0C0C0C
0C0C0C0")
420 CALL CHAR(128,"FFC0C0C0C
0C0C0C0")
430 CALL CHAR(129,"FF8080808
0808080")
440 CALL CHAR(130,"FF8383838
3838383")
450 CALL CHAR(91,"00")
460 CALL CHAR(140,"00")
470 CALL CHAR(148,"00")
480 CALL CHAR(136,"FFFFFFFFF
FFFFFFF")
```

NOTE [=FCTN R

```
490 CALL CHAR(144,"FFFFFFFFF
FFFFFFF")
500 CALL CHAR(137,"C0C0C0C0C
0C0C0C0")
510 CALL CHAR(138,"F0F0F0F0F
0F0F0F0")
520 CALL CHAR(139,"FCFCFCFCF
CFCFCFC")
530 CALL CHAR(145,"C0C0C0C0C
0C0C0C0")
540 CALL CHAR(146,"F0F0F0F0F
0F0F0F0")
550 CALL CHAR(147,"FCFCFCFCF
CFCFCFC")
560 CALL COLOR(13,15,16)
570 CALL COLOR(14,3,16)
580 CALL COLOR(15,7,16)
590 RETURN
600 REM  OPEN FILE/GET DATA
610 CALL CLEAR
620 PRINT "LOAD LATEST BUDGE
T CASSETTE"::::TAB(9);"HIT A
NY KEY"::::
630 CALL KEY(3,KY,ST)
640 IF ST=0 THEN 630
650 OPEN #1:"CS1",INTERNAL,I
NPUT ,FIXED 192
660 FOR I=1 TO 28 STEP 9
670 C=-20
680 INPUT #1:X$
690 K=I+8
700 IF K>35 THEN 710 ELSE 72
0
710 K=K-1
720 FOR J=I TO K
730 C=C+21
740 A$(J)=SEG$(X$,C,8)
750 B$(J)=SEG$(X$,C+8,5)
760 C$(J)=SEG$(X$,C+13,8)
770 IF VAL(B$(J))<>0 THEN 79
0
780 IF VAL(C$(J))<>0 THEN 79
0 ELSE 800
790 A(J)=1
800 NEXT J
810 NEXT I
820 SR$=SEG$(X$,169,6)
830 BB$=SEG$(X$,175,10)
```

```
840 LT$=SEG$(X$,185,2)
850 CLOSE #1
860 CALL CLEAR
870 PRINT "START DATE: ";SR$
:::
880 INPUT "AS OF DATE: ":ND$
890 RETURN
900 REM  DISPLAY ACCTS
910 CALL CLEAR
920 CALL SCREEN(4)
930 FOR I=2 TO 8
940 CALL COLOR(I,2,1)
950 NEXT I
960 PRINT TAB(7);"ACTIVE ACC
OUNTS"::::
970 PRINT "  # NAME";TAB(14)
;"  # NAME":::
980 FOR I=1 TO 35
990 IF A(I)<1 THEN 1030
1000 IF I<10 THEN 1010 ELSE
1020
1010 PRINT " ";
1020 PRINT I;A$(I),
1030 NEXT I
1040 PRINT ::"ENTER FIVE ACC
OUNT NUMBERS":"BELOW.  IF YO
U DON'T WANT":"FIVE, ENTER Z
EROS."::
1050 PRINT "EXAMPLE: 1,2,6,7
,0"::
1060 INPUT "NUMBERS? ":B(1),
B(2),B(3),B(4),B(5)
1070 RETURN
1080 REM  CREATES DISPLAY
1090 CALL CLEAR
1100 CALL SCREEN(11)
1110 CALL HCHAR(1,3,38,28)
1120 FOR R=2 TO 22
1130 CALL HCHAR(R,4,129,26)
1140 NEXT R
1150 CALL HCHAR(23,3,38,28)
1160 CALL VCHAR(1,3,38,22)
1170 CALL VCHAR(1,30,38,22)
1180 MSG$="0204[[[[[[[[BUDG
ET[[[AVERAGE["
1190 GOSUB 2560
1200 CALL HCHAR(2,11,136)
1210 CALL HCHAR(2,20,144)
1220 RETURN
```

NOTE [=FCTN R

```
1230 REM  PRINT ACCT,YTD,BDG
1240 FOR I=2 TO 8
1250 CALL COLOR(I,2,16)
1260 NEXT I
1270 K=0
1280 GOSUB 1420
1290 K=0
1300 FOR M=4 TO 20 STEP 4
1310 K=K+1
1320 ROW$=STR$(M)
1330 IF LEN(ROW$)=2 THEN 135
0
1340 ROW$="0"&ROW$
1350 MSG$=ROW$&"05"&AP$(B(K)
)
1360 MSG$=MSG$&CHR$(91)&BP$(
B(K))
1370 MSG$=MSG$&CHR$(91)&CP$(
B(K))
1380 GOSUB 2560
1390 NEXT M
1400 RETURN
1410 REM  REDO STRINGS
1420 FOR I=1 TO 5
1430 T$=""
1440 TT$=""
1450 T$=A$(B(I))
1460 FOR J=1 TO 8
1470 IF SEG$(T$,J,1)=" " THE
N 1500
1480 TT$=TT$&SEG$(T$,J,1)
1490 GOTO 1510
1500 TT$=TT$&CHR$(91)
1510 NEXT J
1520 AP$(B(I))=TT$
1530 NEXT I
1540 FOR I=1 TO 5
1550 T$=""
1560 TT$=""
1570 T$=B$(B(I))
1580 FOR J=1 TO 5
1590 IF SEG$(T$,J,1)=" " THE
N 1620
1600 TT$=TT$&SEG$(T$,J,1)
1610 GOTO 1630
1620 TT$=TT$&CHR$(91)
1630 NEXT J
1640 BP$(B(I))=TT$
1650 NEXT I
```

```
1660 FOR I=1 TO 5
1670 T$=""
1680 TT$=""
1690 T$=STR$(INT(AVP(I)))
1700 IF LEN(T$)=8 THEN 1730
1710 T$=" "&T$
1720 GOTO 1700
1730 FOR J=1 TO 8
1740 IF SEG$(T$,J,1)=" " THE
N 1770
1750 TT$=TT$&SEG$(T$,J,1)
1760 GOTO 1780
1770 TT$=TT$&CHR$(91)
1780 NEXT J
1790 CP$(B(I))=TT$
1800 NEXT I
1810 RETURN
1820 REM  CALCULATES AVERAGE
1830 D1=VAL(SEG$(SR$,1,2))
1840 D2=VAL(SEG$(SR$,3,2))
1850 D3=VAL(SEG$(SR$,6,1))
1860 D4=VAL(SEG$(ND$,1,2))
1870 D5=VAL(SEG$(ND$,3,2))
1880 D6=VAL(SEG$(ND$,6,1))
1890 DF1=(D4-D1)*30
1900 DF2=D5-D2
1910 DF=DF1+DF2+1
1920 DF=DF/30
1930 FOR I=1 TO 5
1940 J=B(I)
1950 IF C$(J)="" THEN 1960 E
LSE 1970
1960 C$(J)="0"
1970 AVP(I)=VAL(C$(J))/DF
1980 NEXT I
1990 RETURN
2000 REM  CALC INCREMENT
2010 BG=0
2020 INC=0
2030 FOR I=1 TO 5
2040 IF AVP(I)>BG THEN 2110
2050 IF B$(B(I))="" THEN 206
0 ELSE 2070
2060 B$(B(I))="0"
2070 IF VAL(B$(B(I)))>BG THE
N 2090
2080 GOTO 2130
2090 BG=VAL(B$(B(I)))
2100 GOTO 2130
2110 BG=AVP(I)
2120 GOTO 2050
2130 NEXT I
2140 INC=BG/20
2150 RETURN
2160 REM  PRINTS LINES
2170 RW=1
2180 FOR I=1 TO 5
2190 J=B(I)
2200 L1=VAL(B$(J))/INC
2210 IF L1-INT(L1)=0 THEN 22
70
2220 IF L1-INT(L1)<.25 THEN
2290
2230 IF L1-INT(L1)<.50 THEN
2310
2240 IF L1-INT(L1)<.75 THEN
2330
2250 L3=136
2260 GOTO 2340
2270 L3=140
2280 GOTO 2340
2290 L3=137
2300 GOTO 2340
2310 L3=138
2320 GOTO 2340
2330 L3=139
2340 L2=AVP(I)/INC
2350 IF L2-INT(L2)=0 THEN 24
10
2360 IF L2-INT(L2)<.25 THEN
2430
2370 IF L2-INT(L2)<.50 THEN
2450
2380 IF L2-INT(L2)<.75 THEN
2470
2390 L4=144
2400 GOTO 2480
2410 L4=148
2420 GOTO 2480
2430 L4=145
2440 GOTO 2480
2450 L4=146
2460 GOTO 2480
2470 L4=147
2480 RW=RW+4
2490 CALL HCHAR(RW,4,136,L1)
2500 CALL HCHAR(RW,L1+4,L3)
```

```
2510 CALL HCHAR(RW+1,4,144,L
2)
2520 CALL HCHAR(RW+1,L2+4,L4
)
2530 NEXT I
2540 RETURN
2550 REM  MESSAGE ROUTINE
2560 MSGL=LEN(MSG$)
2570 R=VAL(SEG$(MSG$,1,2))
2580 C=VAL(SEG$(MSG$,3,2))
2590 C=C-1
2600 NMSG$=SEG$(MSG$,5,MSGL-
4)
2610 FOR I=5 TO MSGL
2620 C=C+1
2630 CALL HCHAR(R,C,ASC(SEG$
(MSG$,I,1)))
2640 NEXT I
2650 RETURN
```

HAPPY COMPUTING!

CHAPTER SEVEN

Arrays

GENERAL. Some things just go together like "Ham & Eggs" and "Bread & Butter". So it is with ARRAYS and the FOR - NEXT statement. It's rarely possible to build any kind of a meaningful program without using both of these statements in concert with each other. The power of the subscripted variable such as A(I), simply can't be overstated. You have an application for arrays anytime you have a series of numbers or things that you're going to have to keep track of, such as: players on a team; game scores; names & addresses; or monthly sales figures. Before going further in this chapter, as a review it would be worthwhile to reread the sections in your users' manual regarding Arrays, the DIMension statement, and the OPTION BASE statements. Having done this, we too are going to go over some of the important points about arrays that you should be aware of.

Subscripted Variable. All variables used in arrays are known as "subscripted variables". By this we mean that it's any normal variable name, such as A, AMT, TOTAL, etc., followed by some number, or variable representing a number, enclosed within parenthesis, so that the variable looks like: A(2); AMT(I); or TOTAL(7). The number within parenthesis tells the computer which "element" in the array you are referring to. In its simplest form, the creation and printing of a one dimensional array takes the following form:

```
>100 DATA 3,7,6,1
>110 FOR I=1 TO 4
>120 READ A
>130 NR(I)=A
>140 NEXT I
>150 FOR I=1 TO 4
>160 PRINT NR(I)
>170 NEXT I
>RUN
```

The above program simply takes each of the numbers in the data statement and sets it equal to the variable NR. The only thing that distinquishes between the different variables named NR is the subscript that follows the variable. When variables are named in this fashion they are said to be "elements of an array", i.e. element 1, NR(1), is equal to 3; element 3, NR(3), is equal to 6. The beauty of the array and the FOR - NEXT statement is that you can increase the number of elements to almost any level (provided you don't run out of memory), as well as assign and manipulate these numbers, with a limited number of program lines. For instance, we could add 6 more numbers to the data statement in line 100 above; change the number 4 in lines 110 and 150 to a number 10; and the program would create 10 elements as easily, and with the same number of program lines, as it created 4 variables.

DIMension. Some programs which utilize arrays require the use of a dimension statement and some do not. You'll require a dimension statement only if the number of elements in the array is

going to exceed 1Ø. In the above example it did not, so no dimension statement was used. The exact number of elements or the maximum number of elements to be permitted in an array will have to be determined at the beginning of the program and the appropriate dimension statement will have to express this value. Once dimensioned, you cannot increase or decrease this value during the running of the program, nor can you dimension something utilizing a variable created in the running of a program. The rules mentioned in the last two sentences, while perhaps confusing, are extremely important and the following examples will illustrate these points.

```
>1ØØ CALL CLEAR
>11Ø INPUT "A NUMBER:  ":A
>12Ø I=I+1
>13Ø B(I)=A
>14Ø GOTO 11Ø
>RUN
```

The above is one example of a method of inputting a series of numbers, incrementing a subscripted variable, and assigning that number as an element of an array. Enter the above and begin entering numbers (use any numbers you desire). The program keeps accepting numbers and returning to the input statement until the value of I reaches 11. When the computer tries to create an element called B(11), you receive the error message * BAD SUBSCRIPT IN 13Ø *. Add the following line to the program and RUN it again. With this line added you'll be able to enter 14 numbers and the program will error out on the 15th element.

```
>125 DIM B(14)
```

Logic might dictate that the answer to this problem is to change line 125 to ">125 DIM B(I)". If you attempt this, you'll get the error message * INCORRECT STATEMENT IN 125 *. You cannot use a variable for the subscript in the dimension statement -- the subscript must be a positive integer value (whole number). The following might appear to be a way to increase the value of the array in increments based on the number of inputs. Attempting to run this program will result in an immediate error message * NAME CONFLICT IN 16Ø *, demonstrating that you cannot change a dimension once stated.

```
>1ØØ CALL CLEAR
>11Ø INPUT "A NUMBER:  ":A
>12Ø I=I+1
>13Ø IF I<15 THEN 16Ø
>14Ø DIM B(25)
>15Ø GOTO 17Ø
>16Ø DIM B(15)
>17Ø B(I)=A
>18Ø GOTO 11Ø
>RUN
```

Now, in spite of the fact that the first program shown above would adequately work for less than 1Ø elements, the proper way to set up an array and provide for input would be as follows:

```
>1ØØ DIM B(2Ø)
>11Ø CALL CLEAR
>12Ø FOR I=1 TO 2Ø
>13Ø INPUT "A NUMBER (OR 99):
 ":A
>14Ø IF A=99 THEN 17Ø
>15Ø B(I)=A
>16Ø NEXT I
>17Ø STOP
>RUN
```

In this example the subscript in the DIMension statement agrees with the maximum number in the FOR - NEXT loop. We also added another feature which permits you to terminate the FOR - NEXT loop by entering the number "99". With this type of arrangement, you can never get into a situation where the program is trying to assign a subscript to a variable that is higher than your dimension statement permits.

OPTION BASE. The fact that this statement exists at all causes a great deal of concern for beginners. In reality, it seldom will affect your program whether you use it or not. In the above example, as the program is written, where we DIMension B(2Ø), we actually could input 21 numbers by changing the FOR - NEXT statement to read FOR I=Ø TO 2Ø. The extra element would be B(Ø) and can be referenced just like any other element in the array. By putting in the OPTION BASE 1 statement we would eliminate this possibility. If we don't put in the OPTION BASE 1 statement, the only difference is that the computer will automatically reserve 8 bytes of memory for the variable B(Ø). In this case, adding the additional line will consume more than 8 bytes, so it is more "memory efficient" to leave it out. When we get into larger arrays, and multiple dimension arrays, which we'll discuss later, this can make a significant difference.

For most FOR - NEXT applications it is usually more convenient to start numbering your elements with 1 as opposed to zero. If you have the scores of 2Ø rounds of golf stored in an array, the 14th score would also be the 14th element, not the 13th, which it would be if you were using the subscript (Ø). There are times when it is convenient to use the first element (Ø) as a type of "header" in the same way we talked about headers in data files. In this spot you might store something like the total of all items in the array (if they are numbers), or the total number of items in the array (for a list of names).

String Arrays. The most obvious use for string arrays is where you need to store alpha type information such as names and addresses; however, there is no reason why it can't also be used to store numeric information and in many cases it's by far the best method available. They do make reference to the string array in your user's manual; however, they provide practically no examples of its use. If you have entered even a few of our programs to this point, you realize that we use it extensively because of its efficiency from a memory standpoint. Everything we have said previously about DIM statements, OPTION BASE, etc., applies equally to string arrays, the only difference being that the variable name is followed by a dollar sign ($), i.e. A$(I), AMT$(I), or TOTAL$(I).

One, Two & Three Dimensions. This concept seems to bother some people; however, once you get the relationship in mind, it really isn't such a unique treatment of information. We can compare the computer terminology of referencing arrays, i.e. something like A(I,J,K), to either a written outline or in spacial form (length, width, and depth).

RENTAL

```
I. Property 1
   A. Unit 1
      1. Jones, Tom
      2. 1Ø1283
```

B. Unit 2
1. Smith, Kenneth
2. Ø11984

II. Property 2
A. Unit 1
1. Harris, Bill
2. Ø42Ø84
B. Unit 2
1. Jackson, John
2. 12Ø183

The previous example is a typical topic outline which might keep track of two pieces of rental property such as two duplexes. For each unit in each duplex we've recorded the name of the person renting the unit and the expiration date of his lease. The expiration date for Property 2, Unit 1, can be found at II.A.2. In computer format this could be set up as a "three dimensional string array" called RENTAL$(2,2,2). The first digit following the name of the array is the number of main elements (roman numerals); the second digit is the number of sub elements indicated with letters (A,B); and the third digit is the number of secondary sub elements (1,2). To find out the expiration date on Unit 1, Property 2, we would tell the computer to print RENTAL$(2,1,2). It should be noted that the computer array is really only storing 8 pieces of information (2 X 2 X 2), and that is the information shown as 1 and 2 under each of the A's and B's. You could not, for instance, tell the computer to print RENTAL$(1,2). This would result in an error message since the array was originally set up as having three dimensions and three numbers must follow every reference to the array.

There are really only a couple of differences between an outline and the computer method of recording this type of information. First, the computer only uses numbers to reference the information, not roman numerals or letters. Second, and this is important, if you have two sub elements to one main element, then each main element must have exactly two sub elements. For instance, in the above example, if one of the properties had three rental units, then we would have to have a sub element called "C" for both Property I and II. Each of the "C" elements would likewise have to have a 1 and 2 following it. The array would then be called RENTAL$(2,3,2).

If we had property in two cities (Atlanta & Dallas) and we simply wanted to store the addresses of three units in one city and two in the other we might set up a two dimensional array called RENTAL$(2,3). In outline format, the comparable listing would be:

I. Atlanta, GA
A. 134Ø Smithtown Rd
B. 489Ø Harris Ave
C. 172Ø John Street
II. Dallas, TX
A. 222 Houston St
B. None
C. None.

A one dimensional array might just record a series of address or names.

Since everyone is familiar with the game of TIC-TAC-TOE, let's use a version of that game to show the space relationship of three dimensions.

```
( )- - - - ( ) - - - -( )
 |\                   |\
 | ( )        ( )     | ( )
 |   \                |   \
 |    (1)- - - - ( ) - + - -( )
 |     |              |     |
( )- - + - ( ) - - - -( )   |
 |\    |              |\    |
 | ( ) |      (2)     | ( ) |
 |   \ |              |   \ |
 |    ( )- - - - ( ) - + - -( )
 |     |              |     |
( )- - + - ( ) - - - -(3)   |
  \    |                \   |
   ( ) |      ( )        ( )|
     \ |                   \|
      ( )- - - - ( ) - - - -( )
```

The above is a perspective view of a three dimensional game board. As each player makes his move he would put either a "1" or a "Ø" into each of the open spots. If we call this array TTT(3,3,3), the spot currently marked with a "1" is TTT(1,1,1). The spot marked with a "3" is three down, three across, and 3 back, or TTT(3,3,3). The spot marked with a "2" is two down, two across, and 2 back, or TTT(2,2,2). These are generally known as your I, J, and K dimensions: "I" goes from top to bottom; "J" goes from left to right; and "K" goes from front to back. Included with this manual there is a 3-D TIC-TAC-TOE game which actually has four positions in each direction. This program has been purposely set up to ask for I, J, and K to make each move. It's a short program and worth putting in for the practice in referencing various points in an array.

Arrays & Memory.

It may seem that we spend an inordinate amount of time discussing this problem of memory; however, when you begin to fill up arrays with data you'll also begin to appreciate how important it is to conserve every byte possible.

To this point, in our comparisons between numerical and string arrays, we have generalized in saying that a string array consumes no memory until it is actually filled with data. This statement is not precisely so. The following general program was set up to see the effect on memory of creating 1, 2, and 3 dimensional string and numerical arrays.

```
100 OPTION BASE 1
110 DIM A(5,5,5)
120 X=X+8
130 GOSUB 120
```

We substituted various values for the DIM statement and let the program error out on the memory check. For comparison purposes here are the results of our tests:

Numerical Array A with following DIM's

2 - 14488	2,2 - 14464	2,2,2 - 14424
3 - 14480	3,3 - 14424	3,3,3 - 14272
4 - 14472	4,4 - 14368	4,4,4 - 13976
5 - 14464	5,5 - 14296	5,5,5 - 13488
16- 14376		
25- 14304		
64- 13992		
125-13504		

String Array A$ with following DIM's

2 - 14496	2,2 - 14488	2,2,2 - 14472
3 - 14496	3,3 - 14480	3,3,3 - 14432
4 - 14496	4,4 - 14464	4,4,4 - 14360
5 - 14488	5,5 - 14448	5,5,5 - 14240
16- 14472		
25- 14448		
64- 14376		
125-14248		

We're not going to walk you through every calculation, but let's discuss the results. Looking at the numerical

array, you'll note that for every increase of 1 in the one dimensional array (first column) we lose 8 bytes of memory. A 5 X 5 array reserves 25 cells for information. It takes 8 bytes more to reserve 25 cells using a two dimensional array as opposed to one dimension. A 4 X 4 X 4 array reserves 64 cells for information. It takes 16 bytes more to reserve 64 cells using a three dimensional array as opposed to one dimension. In the string array example, although it is not quite as clear, you lose 8 bytes of memory for every four cells of information reserved. It also costs you about 2 bytes to go from one dimension to two, and from two to three. Without filling the array at all, it takes approximately 6 bytes more per cell for numeric data than for string data. In many of our programs we use string arrays to store numeric information. In certain instances this is done to conserve memory and, in other cases, it's done to make printing to the screen easier.

The previous tables show the memory remaining after the dimension statement but prior to entering any data. Based on this alone it would seem that string arrays are more efficient; however, for a one dimensional array this is not always the case.

```
>100 OPTION BASE 1
>110 DIM A(500)
>120 FOR I=1 TO 500
>130 A(I)=10
>140 NEXT I
>150 X=X+8
>160 GOSUB 150
>RUN
```

Using the previous sample program we filled a one dimensional string array with 500 ones, i.e. A$(I)="1". Memory remaining after filling it was 14944. We then did the same thing filling a numeric array with 500 ones, i.e. A(I)=1. Memory remaining after this was 10448. Changing the number from 1 to 10 resulted in figures of 10440 for a string array and 10448 for a numeric array. The lesson to be learned is "unless you are working with a single digit number, it is more memory efficient to store numbers in a numeric array than a string array, if it is a one dimensional array".

Where the string value becomes far more efficient is when you have an application for multidimensional arrays. Our samples above only went to a 5 X 5 X 5 dimension. In reality you'll often be working with much larger dimensions. Our Bowling Stats program stores 3 game scores for 6 bowlers for 38 weeks. In addition, it needed two other numbers stored for each player, each week, indicating wins and losses for a "Kitty". If we called this array "SCORE", it would be dimensioned SCORE(6,38,5). Enter the following:

```
>100 DIM SCORE(6,38,5)
>110 X=X+8
>120 GOSUB 110
>RUN
```

Memory remaining after it errors out is 1392 bytes. Obviously, we simply don't have sufficient memory remaining after the dimension statement to build a program to manipulate the data. To get around this problem we could build a two dimensional string array, each element of which contains five numerical facts. Our new array could be called SCORE$(6,38). We can then fill our array with empty elements that look like " 0 0 000". The first three groups of 2 spaces and a zero represent the 3 games and the

last two zeros the other numerical information we want to store. The following program fills this array and then checks remaining memory.

```
>1ØØ DIM SCORE$(6,38)
>11Ø FOR I=1 TO 6
>12Ø FOR K=1 TO 38
>13Ø SCORE$(I,K)="  Ø  Ø  ØØØ"
>14Ø NEXT K
>15Ø NEXT I
>16Ø X=X+8
>17Ø GOSUB 16Ø
>RUN
```

Checking memory after this run indicates that we have stored all data and still have 1Ø424 bytes of memory to build a program. In this example we used a two dimensional array to represent a 3 dimensional array. The same type of savings can be accomplished by using a one dimensional array to represent a two dimensional array.

<u>Manipulating Data.</u> Now let's discuss how we use the FOR - NEXT and nested FOR - NEXT loops to calculate and manipulate both string and numeric data. The following two programs both store 36Ø randomly selected numbers between 1Ø and 2ØØ. The first part of each program puts the number in the array and prints them across the screen in groups of 6 (6Ø lines). The second part of each program goes back through the array and adds each group of six and prints the total for each group. At the end of the program it prints the total for all 36Ø elements and then errors out on memory check. We've put CALL KEY statements in after the first part of each so that the sections can be timed for efficiency. Part of our remaining discussion on arrays will concern the way in which this information is printed to the screen, so it'll be helpful to put each of these in and run them at least once to see the effect. The first program is a numeric array set up as A(5,12,6). Execution time for part 1 of the numeric array is 1 Min 1Ø Sec. Part 2 takes 3Ø seconds, including the memory check. Memory remaining after execution is 11,128 bytes.

```
>1ØØ CALL CLEAR
>11Ø RANDOMIZE
>12Ø OPTION BASE 1
>13Ø DIM A(5,12,6)
>14Ø FOR I=1 TO 5
>15Ø FOR J=1 TO 12
>16Ø FOR K=1 TO 6
>17Ø A(I,J,K)=INT((191*RND)+1Ø)
>18Ø PRINT A(I,J,K);
>19Ø NEXT K
>2ØØ PRINT
>21Ø NEXT J
>22Ø NEXT I
>23Ø PRINT "STOP TIME=HIT KEY"
>24Ø CALL KEY(3,KY,ST)
>25Ø IF ST=Ø THEN 24Ø
>26Ø CALL CLEAR
>27Ø PRINT "START TIME"
>28Ø FOR I=1 TO 5
>29Ø FOR J=1 TO 12
>3ØØ FOR K=1 TO 6
>31Ø T=T+A(I,J,K)
>32Ø NEXT K
>33Ø PRINT T
>34Ø TT=TT+T
>35Ø T=Ø
>36Ø NEXT J
>37Ø NEXT I
>38Ø PRINT TT
>39Ø X=X+8
>4ØØ GOSUB 39Ø
>RUN
```

The second program stores the same type of information in a two dimensional array called A$(5,12). Each element of this array actually contains six numbers, so each string is 18 characters long after it is completed. Execution time for part 1

of the string array is 1 Min 44 Sec. Part 2 takes 51 seconds, including the memory check. Memory remaining after execution is 12,448 bytes.

```
>1ØØ CALL CLEAR
>11Ø RANDOMIZE
>12Ø OPTION BASE 1
>13Ø DIM A$(5,12)
>14Ø FOR I=1 TO 5
>15Ø FOR J=1 TO 12
>16Ø FOR K=1 TO 6
>17Ø Y$=STR$(INT((191*RND)+1Ø))
>18Ø IF LEN(Y$)=3 THEN 21Ø
>19Ø Y$=" "&Y$
>2ØØ GOTO 18Ø
>21Ø T$=T$&Y$
>22Ø PRINT " "&Y$;
>23Ø NEXT K
>24Ø A$(I,J)=T$
>25Ø T$=""
>26Ø PRINT
>27Ø NEXT J
>28Ø NEXT I
>29Ø PRINT "STOP TIME-HIT KEY"
>3ØØ CALL KEY(3,KY,ST)
>31Ø IF ST=Ø THEN 3ØØ
>32Ø CALL CLEAR
>33Ø PRINT "START TIME"
>34Ø FOR I=1 TO 5
>35Ø FOR J=1 TO 12
>36Ø FOR K=1 TO 6
>37Ø T=T+VAL(SEG$(A$(I,J),(K*3)-2,3))
>38Ø NEXT K
>39Ø PRINT T
>4ØØ TT=TT+T
>41Ø T=Ø
>42Ø NEXT J
>43Ø NEXT I
>44Ø PRINT TT
>45Ø X=X+8
>46Ø GOSUB 45Ø
>RUN
```

From the standpoint of construction, both of these programs are really very similar. Both have a major FOR - NEXT loop of I=1 TO 5 and two "nested FOR - NEXT" loops for the J and K values.

The major difference is what happens within the inner loop (the K loop). In the case of a numeric array we can directly assign a value to the array and print the value using the statements shown in line 17Ø & line 18Ø of the first program. In the second program we can't assign anything to the array until it completes the inner K loop. We assign all six numbers at one time in line 24Ø. In the inner loop we first create a string variable using the STR$ command (line 17Ø); we "pad" each variable to insure that each is identical in length and at least as many digits as the highest number (line 18Ø & 19Ø); then we string these six smaller strings together to form a temporary string variable called T$ (line 21Ø). In the second part of the program, where we total these figures, there is again just a slight difference. In the first program, since it is already numeric, we can add numbers directly (line 31Ø). In the second program we use the value of K, the SEG$ command, and VAL command, to "pull out" a three digit section of the 18 character string, and return it to its numeric value. We can then add or perform any other calculations using this number like any other number.

Remember we mentioned the OPTION BASE 1 for multi dimensional arrays. If we just remove line 12Ø from the sample numeric array program, our remaining memory will decrease from 11128 to 9656. This is a loss of 1472 bytes by failing to use OPTION BASE 1. In the string array sample our loss is only 24 bytes. The rule to remember is "on multi dimensional arrays, particularly numeric arrays, either use element "Ø" of the array, or remember to use OPTION BASE 1".

There are two disadvantages of the string method. First, it is more time consuming: 1Ø4 seconds for part 1 of the string array vs 7Ø seconds for the numeric array; 51 seconds for part 2 of the string array against 3Ø seconds for the numeric version. Second, the programming is slightly more complex.

There are also two advantages to the string method. First, we store 36Ø numbers using 1,32Ø bytes less of memory. In a larger array, or an array of single digit numbers, this difference would be even greater. Still, this amounts to a savings of 3.67 bytes per item stored. Second, the ability to print neatly to the screen is greatly enhanced. In spite of the fact that the numeric program is printing one number right after another, using the semicolon command in line 18Ø, you still wind up with 2 spaces between each number. If you happen to get six, 3 digit, numbers on a single line, they "roll over" to the next line. Using the TAB command you could get a maximum of 5 columns of information across the screen; however, if you left it in numeric form, each column would still be left justified, instead of right justified, as we would like to see it. In the string version, our "numbers" are already right justified and all 3 digits long. We even have to add a space to get separation (line 22Ø of program 2). Using this method we could very nicely print 7 columns across the screen.

As we've said many times, everything is a "trade-off" in programming. If you have a program that requires a lot of screen displays, like our bowling, baseball, or checkbook program, you'll probably need to turn your numbers into strings at some point to get good displays. Why not store them that way in the first place? On the other hand, if you're working with a scientific "number cruncher" type program, that's relatively short in terms of program length, but long in terms of calculations, then numeric arrays and speed are the best choice. Many programs wind up with a combination of both.

Miscellaneous. Just a couple more thoughts on this topic before we move on to searching and sorting arrays. The first concerns how much information you put in an array and this principle also holds true for data files. Store only raw data. In the bowling program, we don't use the array to store the total for each 3 games series. If we want to print that information to the screen, or to a printer, it can be calculated and printed at exactly the time needed. In most cases, there is no need to use up precious memory storing totals. There are exceptions to this, such as the bank balance and YTD totals which we carry forward in the Budget Maintenance program.

Second, don't add an extra dimension to a multi dimensional array to store something like a title or name. In the bowling or baseball stats program we could add an extra dimension to the string and store each players name; however, in the bowling program we would wind up storing each of the six names 38 times. Set up the names of individuals or account names in a one dimensional array so that the "I" value (or whatever other variable you use) corresponds to the first value in the multi dimensional array. If you do this you can reference it any time as you go though your FOR - NEXT or nested loops.

Be creative with the use of arrays and experiment with what they can do. We've given you just a few examples here of how to test various combinations. Look at some of our programs and try to figure out why we did what we did. Maybe you'll even discover a better method. In the Kamakaze game we use two arrays to keep track of the positions of the various planes. In the Patience Please game we used one array, CODE$(83), that was dimensioned much larger than what we needed and we really only used four elements of it. We arranged it so that the subscript would correspond to the ASCII character code number for certain letters. In the TIC-TAC-TOC-TOE game, the screen display is like a 4 X 4 X 4 array, but internally we use a 1Ø X 1Ø, two dimensional array.

Although this ends the chapter on arrays, the next chapter, which deals with Sorting, could really be considered an extension of this topic since both of these normally involve the array and FOR - NEXT loops.

```
*******************
*   BOWLING STATS  *
*    V-PK631KB     *
**   BY T CASTLE   *
*******************
```

DESCRIPTION. This is for all of you "Keglers" out there. It really doesn't require much of a description, because it's just your basic team record keeping system. Your program begins with the MAIN MENU which has all of the usual options for a data file program. Start your season by selecting No. 5, which permits you to build your main roster. We've allowed for 6 names, since you may lose one person and pick up another during the year. The main purpose of the file is to keep track of "individual" scores, not team totals, so you need not enter subs. After you enter and verify your starting roster, the program will instruct you to put in a blank cassette. The program then builds an "empty data" file containing sufficient room for 6 players, 3 games per series, and a total of 38 series. You may also record how many opportunities a person had to win a "kitty" and how many they actually won. You may wish to use these for some other purpose.

Option 2 on the Main Menu is used both for inputting each weeks scores and/or for correcting previous scores. Although this is rare, when building a program of this type, you really must allow for the possibility. All you need to enter are the three game scores. After entering the three games the computer displays the series total and average for that series. You must verify this information, by answering with a "Y".

Option 3 on the Main Menu sends you to a sub menu for display options. After you have just loaded your data, or anytime you have added additional data since using the display option, you'll have to recalculate all of the totals. You do this by answering "Y" to the first question asked on the display option. Your choices for the Display Menu are shown in lines 1410-1450 of the program.

NOTES. The information which must be brought into memory and stored in arrays in this program exceeds the amount in the Baseball Stats program. In order to keep the size of the program down (in terms of lines of code) a couple of nice features have been omitted which would have made it operate somewhat faster. In series 1 as well as series 38, 18 lines of data are always stored on cassette and must be read in prior to updating. If you're ambitious, and if your season runs less than 38 weeks, you could gain extra memory by changing the dimension statements and all other statements that set the limit to 38 series. You would also have to modify the input and output loops to reflect the proper number of "blanks". To further speed the operation you could then place some type of "end of file" indicator in the data record, so that the computer would not have to read in any more data than is necessary.

```
100 REM  *****************
110 REM  * BOWLING STATS *
120 REM  *****************
130 REM
140 REM  BY T CASTLE
150 REM  AMLIST V-PK631KB
160 REM
170 REM  INITIAL DATA
180 GOSUB 32Ø
190 REM  MENU
200 CALL CLEAR
210 PRINT TAB(7);"MAIN MENU"
::::
220 PRINT " 1. INPUT PREVIOU
S DATA"
230 PRINT " 2. ENTER NEW OR
CHANGE DATA"
240 PRINT " 3. DISPLAY OPTIO
NS"
250 PRINT " 4. SAVE DATA"
260 PRINT " 5. START SEASON"
::::::::::
270 INPUT "OPTION? ":Q$
280 IF (ASC(Q$)<49)+(ASC(Q$)
>53)THEN 27Ø
290 ON VAL(Q$)GOSUB 44Ø,243Ø
,13ØØ,227Ø,344Ø
300 GOTO 2ØØ
310 REM  INITIAL DATA
320 DIM BW$(6,38)
330 DIM NM$(6)
340 DEF SER=((J-1)/11)+AD
350 DEF STT=G1+G2+G3
360 DEF SAV=INT((STT/3)+.5)
370 DEF TT=VAL(SEG$(BW$(I,K)
,1Ø,1))
380 DEF WT=VAL(SEG$(BW$(I,K)
,11,1))
390 DEF G1T=VAL(SEG$(BW$(I,K
),1,3))
400 DEF G2T=VAL(SEG$(BW$(I,K
),4,3))
410 DEF G3T=VAL(SEG$(BW$(I,K
),7,3))
420 RETURN
430 REM  FILE INPUT
440 CALL CLEAR
450 PRINT "REMOVE PROGRAM CA
SSETTE-PUT"
460 PRINT "IN PREVIOUS DATA
CASSETTE":::
470 PRINT "          HIT ANY K
EY"::::::
480 CALL KEY(3,KY,ST)
490 IF ST=Ø THEN 48Ø
500 OPEN #1:"CS1",INTERNAL,I
NPUT ,FIXED 192
510 FOR I=1 TO 6
520 FOR AD=Ø TO 26 STEP 13
530 INPUT #1:X$
540 FOR J=1 TO 133 STEP 11
550 IF (AD>Ø)+(J>1)THEN 58Ø
560 NM$(I)=SEG$(X$,1,1Ø)
570 GOTO 59Ø
580 BW$(I,SER)=SEG$(X$,J,11)
590 NEXT J
600 NEXT AD
610 NEXT I
620 CLOSE #1
630 X$=""
640 RETURN
650 REM  MAIN CALCULATIONS
660 FOR I=1 TO 6
670 CALL CLEAR
680 PRINT "CALCULATING TOTAL
S ";NM$(I)
690 FOR K=1 TO 38
700 IF BW$(I,K)="  Ø  Ø  ØØØ
" THEN 95Ø
710 TOT(I)=TOT(I)+TT
720 TOW(I)=TOW(I)+WT
730 IF G1T=Ø THEN 79Ø
740 PTOT(I)=PTOT(I)+G1T
750 GTOT(I)=GTOT(I)+1
760 IF G1T<TGS(5)THEN 79Ø
770 GAME=G1T
780 GOSUB 114Ø
790 IF G2T=Ø THEN 85Ø
800 PTOT(I)=PTOT(I)+G2T
810 GTOT(I)=GTOT(I)+1
820 IF G2T<TGS(5)THEN 85Ø
830 GAME=G2T
840 GOSUB 114Ø
850 IF G3T=Ø THEN 91Ø
860 PTOT(I)=PTOT(I)+G3T
870 GTOT(I)=GTOT(I)+1
880 IF G3T<TGS(5)THEN 91Ø
890 GAME=G3T
900 GOSUB 114Ø
910 IF GTOT(I)=Ø THEN 95Ø
920 ATOT(I)=INT((PTOT(I)/GTO
T(I))+.5)
```

```
930 IF G1T+G2T+G3T<TSS(5)THE
N 950
940 GOSUB 990
950 NEXT K
960 NEXT I
970 RETURN
980 REM  CHANGES HIGH SERIES
990 FOR FND=1 TO 5
1000 IF G1T+G2T+G3T<TSS(FND)
THEN 1030
1010 CHG=FND
1020 FND=5
1030 NEXT FND
1040 FOR FND=5 TO CHG+1 STEP
 -1
1050 TSS(FND)=TSS(FND-1)
1060 TSN(FND)=TSN(FND-1)
1070 TSE(FND)=TSE(FND-1)
1080 NEXT FND
1090 TSS(CHG)=G1T+G2T+G3T
1100 TSN(CHG)=I
1110 TSE(CHG)=K
1120 RETURN
1130 REM  CHANGES HIGH GAME
1140 FOR FND=1 TO 5
1150 IF GAME<TGS(FND)THEN 11
90
1160 CHG=FND
1170 FND=5
1180 TGN(FND)=TGN(FND-1)
1190 NEXT FND
1200 FOR FND=5 TO CHG+1 STEP
 -1
1210 TGS(FND)=TGS(FND-1)
1220 TGN(FND)=TGN(FND-1)
1230 TGE(FND)=TGE(FND-1)
1240 NEXT FND
1250 TGS(CHG)=GAME
1260 TGN(CHG)=I
1270 TGE(CHG)=K
1280 RETURN
1290 REM  INDIVIDUAL DISPLAY
1300 CALL CLEAR
1310 PRINT "ANSWER (Y) BELOW
 IF NEW DATA"
1320 PRINT "HAS  BEEN  ADDED
 OR  IF DATA"
1330 PRINT "HAS JUST BEEN LO
ADED"::::::::
1340 INPUT "NEW CALCULATIONS
(Y OR N)? ":Q$
1350 IF Q$="N" THEN 1380
1360 IF Q$="Y" THEN 1370 ELS
E 1340
1370 GOSUB 660
1380 REM  DISPLAY MENU
1390 CALL CLEAR
1400 PRINT TAB(7);"DISPLAY O
PTIONS"::::
1410 PRINT " 1. ANY SERIES,
ALL PLAYERS"
1420 PRINT " 2. ANY PLAYER,
ALL GAMES"
1430 PRINT " 3. HIGH GAMES/S
ERIES"
1440 PRINT " 4. BASIC TEAM S
TATS"
1450 PRINT " 5. RETURN TO ME
NU"::::::::
1460 INPUT " SELECTION? ":Q$
1470 IF LEN(Q$)<>1 THEN 1460
1480 CK=ASC(Q$)
1490 IF (CK<49)+(CK>53)THEN
1460
1500 Q=VAL(Q$)
1510 IF Q=5 THEN 1540
1520 ON Q GOSUB 1560,1770,19
80,2150
1530 GOTO 1390
1540 RETURN
1550 REM  TEAM DISPLAY
1560 CALL CLEAR
1570 PRINT "ENTER SERIES # O
R X FOR MENU"::
1580 INPUT "SELECTION? ":Q$
1590 IF Q$="X" THEN 1750
1600 GOSUB 2990
1610 IF Q$="X" THEN 1570
1620 K=VAL(Q$)
1630 CALL CLEAR
1640 PRINT TAB(5);"SCORES -
SERIES ";K::::
1650 PRINT "NAME     T W GM1
 GM2 GM3 TOT"::
1660 FOR I=1 TO 6
1670 G4T=G1T+G2T+G3T
1680 PRINT SEG$(NM$(I),1,8);
1690 PRINT TAB(10);STR$(TT);
TAB(12);STR$(WT);TAB(14);STR
$(G1T);TAB(18);STR$(G2T);TAB
(22);STR$(G3T);TAB(25);G4T
1700 NEXT I
```

```
1710 PRINT ::TAB(10);"HIT AN
Y KEY"::::
1720 CALL KEY(3,KY,ST)
1730 IF ST=0 THEN 1720
1740 GOTO 1560
1750 RETURN
1760 REM  PLAYER/ALL GAMES
1770 CALL CLEAR
1780 PRINT "ENTER PLAYER # O
R X FOR MENU"::
1790 K=0
1800 INPUT "SELECTION? ":Q$
1810 IF Q$="X" THEN 1960
1820 I=VAL(Q$)
1830 CALL CLEAR
1840 PRINT "PLAYER # ";I;NM$
(I)
1850 PRINT "S#  T  W  GM-1 G
M-2 GM-3 TOT"
1860 K=K+1
1870 IF K=39 THEN 1910
1880 PRINT STR$(K);TAB(4);TT
;TAB(7);WT;TAB(10);G1T;TAB(1
5);G2T;TAB(20);G3T;TAB(25);G
1T+G2T+G3T
1890 IF K=19 THEN 1910
1900 GOTO 1860
1910 PRINT ::"HIT ANY KEY";
1920 CALL KEY(3,KY,ST)
1930 IF ST=0 THEN 1920
1940 IF K=39 THEN 1770
1950 GOTO 1830
1960 RETURN
1970 REM  HIGHS
1980 CALL CLEAR
1990 PRINT TAB(5);"TOP FIVE
SERIES"::
2000 PRINT "PLAYER";TAB(10);
"SERIES";TAB(20);"WEEK"::
2010 FOR I=1 TO 5
2020 PRINT NM$(TSN(I));TSS(I
);TAB(20);TSE(I)
2030 NEXT I
2040 PRINT
2050 PRINT TAB(5);"TOP FIVE
GAMES"::
2060 PRINT "PLAYER";TAB(11);
"GAME";TAB(20);"WEEK"::
2070 FOR I=1 TO 5
2080 PRINT NM$(TGN(I));TGS(I
);TAB(20);TGE(I)
2090 NEXT I
2100 PRINT ::TAB(5);"HIT ANY
 KEY FOR MENU"
2110 CALL KEY(3,KY,ST)
2120 IF ST=0 THEN 2110
2130 RETURN
2140 REM  BASIC TEAM STATS
2150 CALL CLEAR
2160 PRINT TAB(5);"BASIC TEA
M STATS"::
2170 PRINT "NAME    T   W
PINS GMS AVE"::
2180 FOR I=1 TO 6
2190 PRINT SEG$(NM$(I),1,7);
TAB(9);STR$(TOT(I));TAB(13);
STR$(TOW(I));TAB(17);STR$(PT
OT(I));
2200 PRINT TAB(22);STR$(GTOT
(I));TAB(26);STR$(ATOT(I))
2210 NEXT I
2220 PRINT ::;"        HIT ANY
 KEY"::::::
2230 CALL KEY(3,KY,ST)
2240 IF ST=0 THEN 2230
2250 RETURN
2260 REM  FILE OUTPUT
2270 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
2280 FOR I=1 TO 6
2290 FOR K=0 TO 38
2300 IF K>0 THEN 2330
2310 X$=NM$(I)&" "
2320 GOTO 2380
2330 X$=X$&BW$(I,K)
2340 IF (K=12)+(K=25)+(K=38)
THEN 2360
2350 GOTO 2380
2360 PRINT #1:X$
2370 X$=""
2380 NEXT K
2390 NEXT I
2400 CLOSE #1
2410 RETURN
2420 REM  SCREEN INPUT
2430 CALL CLEAR
2440 PRINT "USE TO ENTER WEE
KLY  SCORES"
2450 PRINT "AND CORRECT SCOR
ES"::::::::
```

```
2460 PRINT "       HIT ANY KE
Y"::::::::
2470 CALL KEY(3,KY,ST)
2480 IF ST=0 THEN 2470
2490 CALL CLEAR
2500 PRINT "BOWLING ROSTER":
2510 FOR J=1 TO 3
2520 PRINT J;TAB(4);NM$(J);
2530 PRINT TAB(16);J+3;NM$(J
+3)
2540 NEXT J
2550 PRINT
2560 PRINT "ENTER X FOR MENU
"::
2570 INPUT "SERIES # ":Q$
2580 IF Q$="X" THEN 2970
2590 GOSUB 2990
2600 IF Q$="X" THEN 2570
2610 WK=VAL(Q$)
2620 INPUT "ROSTER # ":Q$
2630 GOSUB 2990
2640 IF Q$="X" THEN 2620
2650 IF VAL(Q$)>6 THEN 2620
2660 I=VAL(Q$)
2670 PRINT
2680 INPUT "GAME 1   ":Q$
2690 GOSUB 3140
2700 IF Q$="X" THEN 2680
2710 G1=VAL(Q$)
2720 INPUT "GAME 2   ":Q$
2730 GOSUB 3140
2740 IF Q$="X" THEN 2720
2750 G2=VAL(Q$)
2760 INPUT "GAME 3   ":Q$
2770 GOSUB 3140
2780 IF Q$="X" THEN 2760
2790 G3=VAL(Q$)
2800 PRINT
2810 PRINT "SERIES ";TAB(9);
STT
2820 PRINT "SER AVE";TAB(9);
SAV:
2830 INPUT "TRYS?    ":TRY$
2840 IF LEN(TRY$)<>1 THEN 28
30
2850 CK=ASC(TRY$)
2860 IF (CK<48)+(CK>57)THEN
2830
2870 INPUT "WINS?    ":WIN$
2880 IF LEN(WIN$)<>1 THEN 28
70
2890 CK=ASC(WIN$)
2900 IF (CK<48)+(CK>57)THEN
2870
2910 PRINT
2920 INPUT "IS THIS CORRECT(
Y OR N)? ":Q$
2930 IF Q$="N" THEN 2490
2940 IF Q$="Y" THEN 2950 ELS
E 2920
2950 GOSUB 3290
2960 GOTO 2490
2970 RETURN
2980 REM  VERIFY SERIES
2990 IF LEN(Q$)=0 THEN 3110
3000 FOR CK=1 TO LEN(Q$)
3010 CK1=ASC(SEG$(Q$,CK,1))
3020 IF (CK1<48)+(CK1>57)THE
N 3040
3030 GOTO 3060
3040 Q$="X"
3050 CK=LEN(Q$)
3060 NEXT CK
3070 IF Q$="X" THEN 3120
3080 CK=VAL(Q$)
3090 IF (CK<1)+(CK>38)THEN 3
110
3100 GOTO 3120
3110 Q$="X"
3120 RETURN
3130 REM  VERIFY GAMES
3140 IF LEN(Q$)=0 THEN 3260
3150 FOR CK=1 TO LEN(Q$)
3160 CK1=ASC(SEG$(Q$,CK,1))
3170 IF (CK1<48)+(CK1>57)THE
N 3190
3180 GOTO 3210
3190 Q$="X"
3200 CK=LEN(Q$)
3210 NEXT CK
3220 IF Q$="X" THEN 3270
3230 CK=VAL(Q$)
3240 IF (CK<0)+(CK>300)THEN
3260
3250 GOTO 3270
3260 Q$="X"
3270 RETURN
3280 REM  CONVERT TO ARRAYS
3290 G1$=STR$(G1)
3300 IF LEN(G1$)=3 THEN 3330
3310 G1$=" "&G1$
3320 GOTO 3300
```

```
3330 G2$=STR$(G2)
3340 IF LEN(G2$)=3 THEN 3370
3350 G2$=" "&G2$
3360 GOTO 3340
3370 G3$=STR$(G3)
3380 IF LEN(G3$)=3 THEN 3410
3390 G3$=" "&G3$
3400 GOTO 3380
3410 BW$(I,WK)=G1$&G2$&G3$&T
RY$&WIN$
3420 RETURN
3430 REM  ONE TIME BUILD
3440 CALL CLEAR
3450 PRINT "USE  THIS  ONLY
 AT START OF"
3460 PRINT "SEASON. MAX OF 6
 NAMES.  HIT"
3470 PRINT "ENTER KEY TO SKI
P NAME"::::::
3480 PRINT "         HIT  ANY
KEY"::::::
3490 CALL KEY(3,KY,ST)
3500 IF ST=0 THEN 3490
3510 CALL CLEAR
3520 FOR I=1 TO 6
3530 PRINT "ROSTER #";I
3540 INPUT "NAME?  ":NM$(I)
3550 IF LEN(NM$(I))>10 THEN
3590
3560 IF LEN(NM$(I))>0 THEN 3
610
3570 NM$(I)="SUB       "
3580 GOTO 3610
3590 PRINT "TOO LONG"
3600 GOTO 3540
3610 PRINT
3620 IF LEN(NM$(I))=10 THEN
3650
3630 NM$(I)=NM$(I)&" "
3640 GOTO 3620
3650 NEXT I
3660 CALL CLEAR
3670 PRINT TAB(8);"TEAM ROST
ER":::
3680 PRINT TAB(3);"ROSTER
    NAME"::
3690 FOR I=1 TO 6
3700 PRINT TAB(4);I;TAB(16);
NM$(I)
3710 NEXT I
3720 PRINT ::::::::
3730 INPUT "IS THIS CORRECT(
Y OR N)? ":Q$
3740 IF Q$="Y" THEN 3770
3750 IF Q$="N" THEN 3440
3760 GOTO 3730
3770 CALL CLEAR
3780 PRINT "REMOVE PROG CASS
ETTE - LOAD"
3790 PRINT "BLANK DATA CASSE
TTE"::::
3800 PRINT "         HIT ANY
KEY"::::::
3810 CALL KEY(3,KY,ST)
3820 IF ST=0 THEN 3810
3830 CALL CLEAR
3840 PRINT "FOLLOW SCREEN IN
STRUCTIONS"::::
3850 ADD$="  0  0  000"
3860 FOR I=1 TO 12
3870 X1$=X1$&ADD$
3880 NEXT I
3890 X2$=X1$&ADD$
3900 X1$=" "&X1$
3910 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
3920 FOR I=1 TO 6
3930 CALL CLEAR
3940 PRINT "PRINTING  DATA F
OR ";NM$(I)
3950 X$=NM$(I)&X1$
3960 PRINT #1:X$
3970 PRINT #1:X2$
3980 PRINT #1:X2$
3990 NEXT I
4000 CLOSE #1
4010 X1$=""
4020 X2$=""
4030 X$=""
4040 RETURN
```

HAPPY COMPUTING!

```
**************************
*    BASEBALL STATS      *
*      V-PF431KB         *
*     BY T CASTLE        *
**************************
```

DESCRIPTION. "Baseball Stats" is a program for all of the avid little league baseball fathers (and mothers) who like to keep track of what their boys (or girls), and the rest of the team, are doing in baseball.

The program begins with the display of a "Main Menu" which permits you to: load previous data; build a roster containing names, player numbers, and positions; input game data for each player including number of hits, walks, strike outs, homeruns, other outs, and reaching base by other means; save data; and call up the display menu. The figures entered in the input section are used to calculate Total At Bats, Official at Bats, Percentage on Base, and Official Average. By the time the main menu is displayed, the program already will have built, and filled with blank information or zeros, variables for all statistics. This program will allow for a maximum of twelve players. For each of the twelve players the program also creates sixteen variables, six digits long. These are defined as a string array, A$(I,K), where "I" is the roster number of the Player and "K" is the game number (1-16). In the beginning, all strings are equal to "ØØØØØØ". For each boy, for each game, each digit from 1 through 6 represents the walks, hits, strike outs, other outs, reached other, and home run figure.

To utilize the program, simply load and RUN. Once the "blank" variables are created, the next thing to do is to build your Roster of Players. This program displays all twelve as R#, NAME, P#, and PS. It then asks for name, player number and position for each of twelve players in sequential order. CAUTION - if you make a mistake the only way to correct it is to reenter all twelve again, so be sure each entry is correct before hitting the "ENTER" key. The program will "pad" each variable to the proper length and will not accept it if it's too long. When the roster is built, the program cycles back to the Main Menu. After the Roster is built, you should load a blank cassette and use the #4 option to SAVE this data. On the next and subsequent use of the program, the first thing you will do is use the #1 option to LOAD previous data. The only time the #2 option for BUILD ROSTER will be used again is at the beginning of a new season.

The number #3 option is used to INPUT GAME DATA for any given player for any given game. Even if you detect an error in your previous data, you can go back to the specific game and player and correct his data. The option asks for Game Number and Roster Number and then displays the Player's Name. Then you must enter a single digit number or zero for WALKS, HITS, STRIKE OUTS, OTHER OUTS, REACHED OTHER, AND HOME RUNS. The program then calculates and displays the TOTAL AB and OFFICIAL AB figure. WALKS and REACHED OTH are not considered Official At Bats. Home Runs do not enter into any calculation and it's simply recorded as a matter of fact. If a player hits a Home Run, it must also be counted as a HIT.

To access the DISPLAY MENU, use option #5. The menu has four options. Option #1 permits you to view all

total data for each of the twelve players. It displays them six at a time in a two line listing. Hitting any key causes the next six players to be displayed. Hitting any key after this causes the program to go back to the display menu. Option #2 displays the raw data for any specified game and player, or for the entire team. Option #3 displays only the selected information which is most commonly used to set lineups. Option #4 returns you to the Main Menu. Prior to each use of the DISPLAY MENU, you will be asked if the data has been calculated. You will need to answer "Y" if you have just loaded data from a cassette and only want to display information or prior to viewing information if you have just entered new data.

The Load and Save options of the main menu have screen displays which walk you through the procedure for using the cassette recorder.

FILE STRUCTURE. As mentioned in Chapter 2, the essence of any functional program is the structure of the data file. This program utilizes a data file 9 lines long, each containing 192 characters. The first line contains the basic information for each player: a name 8 digits long, a number 2 digits long, and position 2 digits long. It then has 12 digits for each player X 12 players or a total of 144 characters. These are strung, one after another, in a single data line and the computer "pads" the line to 192 characters. The second through ninth lines of the data file contain the basic information for each player for two games. This is the value created as A$(I,K). Each A$(I,K) is six digits long and represents the data for one boy for one game. By stringing these values together for one game we have 12 X 6 digits or 72 characters. By adding the data for two games together we create a data line 144 characters long. This is padded to 192. Since there are sixteen games we need eight data lines to store the raw information for all games and all players.

```
100 REM  ******************
110 REM  * BASEBALL STATS *
120 REM  ******************
130 REM  BY T CASTLE
140 REM   AMLIST V-PF431KB
150 REM
160 REM  SET VARIABLES
170 CALL CLEAR
180 DIM N$(12,13)
190 DIM A$(12,16)
200 REM  FILL SETS EMPTY
210 CALL CLEAR
220 FOR I=1 TO 12
230 N$(I,1)="NAME    "
240 N$(I,2)="NR"
250 N$(I,3)="PS"
260 N$(I,12)="0000"
270 N$(I,13)="0000"
280 FOR K=4 TO 11
290 N$(I,K)="00"
300 NEXT K
310 NEXT I
320 FOR I=1 TO 12
330 FOR K=1 TO 16
340 A$(I,K)="000000"
350 NEXT K
360 NEXT I
370 REM   MAIN MENU
380 CALL CLEAR
390 PRINT TAB(8);"MAIN MENU"
:::
400 PRINT TAB(4);"1 - LOAD P
REV DATA"
410 PRINT TAB(4);"2 - BUILD
ROSTER"
420 PRINT TAB(4);"3 - INPUT
GAME DATA"
430 PRINT TAB(4);"4 - SAVE D
ATA"
440 PRINT TAB(4);"5 - DISPLA
Y MENU"::::::
450 PRINT ::::
460 INPUT "SELECTION? ":Q$
470 GOSUB 3930
480 IF Q$="X" THEN 460
490 Q=VAL(Q$)
500 IF Q>5 THEN 460
510 ON Q GOSUB 3530,870,1140
,3200,540
520 GOTO 380
530 REM  DISPLAY OPTIONS
540 CALL CLEAR
550 PRINT "IF YOU HAVE ENTER
ED NEW DATA"
560 PRINT "SINCE YOU USED TH
IS  DISPLAY"
570 PRINT "MENU, YOU WILL HA
VE TO  CAL-"
580 PRINT "CULATE NEW TOTALS
"::::::::
590 PRINT "ENTER Y - NEW CAL
CULATION"
600 PRINT "        N - DISPLAY
 MENU":::
610 INPUT "ANSWER? ":Q$
620 IF Q$="N" THEN 660
630 IF Q$="Y" THEN 650
640 GOTO 610
650 GOSUB 1860
660 CALL CLEAR
670 PRINT TAB(12);"SUB MENU"
::
680 PRINT TAB(9);"DISPLAY OP
TIONS":::
690 PRINT TAB(4);"1 - ALL TO
TAL STATS"
700 PRINT TAB(4);"    ENTIRE
 TEAM"
710 PRINT TAB(4);"2 - RAW TE
AM DATA"
720 PRINT TAB(4);"    ANY GA
ME OR TEAM"
730 PRINT TAB(4);"3 - MOST C
OMMON STATS"
740 PRINT TAB(4);"    ENTIRE
 TEAM"
750 PRINT TAB(4);"4 - MAIN M
ENU"
760 PRINT ::::
770 INPUT "SELECTION? ":Q$
780 GOSUB 3930
790 IF Q$="X" THEN 770
800 Q=VAL(Q$)
810 IF Q=4 THEN 850
820 IF Q>3 THEN 770
830 ON Q GOSUB 2960,2590,244
0
840 GOTO 660
850 RETURN
860 REM  BUILD ROSTER
```

```
870 FOR I=1 TO 12
880 CALL CLEAR
890 PRINT "R# NAME      P# PS
"::
900 FOR IP=1 TO 12
910 PRINT STR$(IP);
920 PRINT TAB(4);N$(IP,1);TA
B(13);
930 PRINT N$(IP,2);TAB(16);N
$(IP,3)
940 NEXT IP
950 PRINT ::"R#";I;" ";
960 INPUT "      NAME? ":N$(I
,1)
970 IF LEN(N$(I,1))=8 THEN 1
010
980 IF LEN(N$(I,1))>8 THEN 9
60
990 N$(I,1)=N$(I,1)&" "
1000 GOTO 970
1010 INPUT "            NR.?
 ":N$(I,2)
1020 IF LEN(N$(I,2))=2 THEN
1060
1030 IF LEN(N$(I,2))>2 THEN
1010
1040 N$(I,2)=N$(I,2)&" "
1050 GOTO 1020
1060 INPUT "            POS?
 ":N$(I,3)
1070 IF LEN(N$(I,3))=2 THEN
1110
1080 IF LEN(N$(I,3))>2 THEN
1060
1090 N$(I,3)=N$(I,3)&" "
1100 GOTO 1070
1110 NEXT I
1120 RETURN
1130 REM  INPUT GAME DATA
1140 CALL CLEAR
1150 INPUT "GAME NO.?  ":Q$
1160 GOSUB 3850
1170 IF Q$="X" THEN 1150
1180 K=VAL(Q$)
1190 IF (K<1)+(K>16)THEN 115
0
1200 INPUT "ROSTER NO.?":Q$
1210 GOSUB 3850
1220 IF Q$="X" THEN 1200
1230 I=VAL(Q$)
1240 IF (I<1)+(I>12)THEN 120
0
1250 PRINT "PLAYER      ";N$(
I,1)::
1260 A1$=""
1270 INPUT "WALKS?      ":Q$
1280 GOSUB 3930
1290 IF Q$="X" THEN 1270
1300 A1$=A1$&Q$
1310 INPUT "HITS?       ":Q$
1320 GOSUB 3930
1330 IF Q$="X" THEN 1310
1340 A1$=A1$&Q$
1350 INPUT "STRIKE OUTS?":Q$
1360 GOSUB 3930
1370 IF Q$="X" THEN 1350
1380 A1$=A1$&Q$
1390 INPUT "OTHER OUTS? ":Q$
1400 GOSUB 3930
1410 IF Q$="X" THEN 1390
1420 A1$=A1$&Q$
1430 INPUT "REACHED OTH?":Q$
1440 GOSUB 3930
1450 IF Q$="X" THEN 1430
1460 A1$=A1$&Q$
1470 INPUT "HOME RUNS?  ":Q$
1480 GOSUB 3930
1490 IF Q$="X" THEN 1470
1500 A1$=A1$&Q$
1510 FOR J=1 TO 5
1520 TB=TB+VAL(SEG$(A1$,J,1)
)
1530 NEXT J
1540 TB$=STR$(TB)
1550 PRINT "TOTAL AB    ";TB
$
1560 FOR J=2 TO 4
1570 AB=AB+VAL(SEG$(A1$,J,1)
)
1580 NEXT J
1590 AB$=STR$(AB)
1600 PRINT "OFFICIAL AB ";AB
$:::
1610 PRINT "ENTER  R - REENT
ER"
1620 PRINT "       N - NEXT
PLAYER"
1630 PRINT "       M - MENU"
::
1640 INPUT "ANSWER? ":Q$
```

```
1650 IF Q$="R" THEN 1690
1660 IF Q$="N" THEN 1750
1670 IF Q$="M" THEN 1810
1680 GOTO 1640
1690 CALL CLEAR
1700 PRINT "GAME ";K
1710 PRINT "ROSTER NO. ";I
1720 TB=0
1730 AB=0
1740 GOTO 1250
1750 CALL CLEAR
1760 TB=0
1770 AB=0
1780 A$(I,K)=A1$
1790 PRINT "GAME ";K
1800 GOTO 1200
1810 TB=0
1820 AB=0
1830 A$(I,K)=A1$
1840 RETURN
1850 REM  ACCUMULATER
1860 CALL CLEAR
1870 FOR I=1 TO 12
1880 CALL CLEAR
1890 PRINT "CALC TOTALS ROST
ER NO.";I:::
1900 FOR J=1 TO 6
1910 FOR K=1 TO 16
1920 ACC=ACC+VAL(SEG$(A$(I,K
),J,1))
1930 NEXT K
1940 ACC$=STR$(ACC)
1950 IF LEN(ACC$)=2 THEN 198
0
1960 ACC$="0"&ACC$
1970 GOTO 1950
1980 N$(I,J+5)=ACC$
1990 ACC=0
2000 NEXT J
2010 TB=0
2020 FOR B=6 TO 10
2030 TB=TB+VAL(N$(I,B))
2040 NEXT B
2050 TB$=STR$(TB)
2060 IF LEN(TB$)=2 THEN 2090
2070 TB$="0"&TB$
2080 GOTO 2060
2090 N$(I,4)=TB$
2100 AB=0
2110 FOR B=7 TO 9
2120 AB=AB+VAL(N$(I,B))
2130 NEXT B
2140 AB$=STR$(AB)
2150 IF LEN(AB$)=2 THEN 2180
2160 AB$="0"&AB$
2170 GOTO 2150
2180 N$(I,5)=AB$
2190 AV=0
2200 IF VAL(N$(I,5))=0 THEN
2230
2210 AV=VAL(N$(I,7))/VAL(N$(
I,5))
2220 AV=INT((AV*1000)+.5)
2230 AV$=STR$(AV)
2240 IF LEN(AV$)=4 THEN 2270
2250 AV$=" "&AV$
2260 GOTO 2240
2270 N$(I,13)=AV$
2280 PB=0
2290 PB1=0
2300 PB1=PB1+VAL(N$(I,6))
2310 PB1=PB1+VAL(N$(I,7))
2320 PB1=PB1+VAL(N$(I,10))
2330 IF VAL(N$(I,4))=0 THEN
2360
2340 PB=PB1/VAL(N$(I,4))
2350 PB=INT((PB*1000)+.5)
2360 PB$=STR$(PB)
2370 IF LEN(PB$)=4 THEN 2400
2380 PB$=" "&PB$
2390 GOTO 2370
2400 N$(I,12)=PB$
2410 NEXT I
2420 RETURN
2430 REM  PRINTS TM DSPLY 3
2440 CALL CLEAR
2450 PRINT "NAME     TB AB H
T  POB  AVE"::
2460 FOR I=1 TO 12
2470 PRINT N$(I,1)&" ";
2480 PRINT N$(I,4)&" ";
2490 PRINT N$(I,5)&" ";
2500 PRINT N$(I,7)&" ";
2510 PRINT N$(I,12)&" ";
2520 PRINT N$(I,13)&" "
2530 NEXT I
2540 PRINT ::"HIT ANY KEY"::
2550 CALL KEY(3,KY,ST)
2560 IF ST=0 THEN 2550
2570 RETURN
2580 REM    TEAM DISPLAY 2
```

```
2590 CALL CLEAR
2600 INPUT "GAME NO. OR T FO
R TEAM ":Q$
2610 IF Q$="T" THEN 2790
2620 GOSUB 3850
2630 IF Q$="X" THEN 2600
2640 Q=VAL(Q$)
2650 IF (Q<1)+(Q>16)THEN 260
0
2660 CALL CLEAR
2670 PRINT TAB(4);"GAME";Q;"
 RAW TEAM DATA"::
2680 PRINT "R# ";
2690 PRINT TAB(4);"NAME     W
K HT SO OO RO HR"::
2700 FOR I=1 TO 12
2710 PRINT STR$(I);
2720 PRINT TAB(4);SEG$(N$(I,
1),1,7);" ";
2730 FOR K=1 TO 5
2740 PRINT " ";SEG$(A$(I,Q),
K,1);" ";
2750 NEXT K
2760 PRINT " ";SEG$(A$(I,Q),
6,1)
2770 NEXT I
2780 GOTO 2910
2790 CALL CLEAR
2800 PRINT TAB(5);"TOTAL RAW
 TEAM DATA"::
2810 PRINT "R# ";
2820 PRINT TAB(4);"NAME     W
K HT SO OO RO HR"::
2830 FOR I=1 TO 12
2840 PRINT STR$(I);
2850 PRINT TAB(4);SEG$(N$(I,
1),1,7);" ";
2860 FOR J=6 TO 10
2870 PRINT N$(I,J);" ";
2880 NEXT J
2890 PRINT N$(I,11)
2900 NEXT I
2910 PRINT ::"HIT ANY KEY"::
2920 CALL KEY(3,KY,ST)
2930 IF ST=0 THEN 2920
2940 RETURN
2950 REM   TEAM DISPLAY 1
2960 I=1
2970 CALL CLEAR
2980 PRINT "R# NAME      NR P
S  AVE"
2990 PRINT "TB AB WK HT SO O
O RO HR  POB"::
3000 FOR J=I TO I+5
3010 PRINT STR$(J);" ";
3020 PRINT TAB(4);N$(J,1);"
";
3030 PRINT N$(J,2);" ";
3040 PRINT N$(J,3);" ";
3050 PRINT N$(J,13)
3060 FOR K=4 TO 11
3070 PRINT N$(J,K);" ";
3080 NEXT K
3090 PRINT N$(J,12)
3100 PRINT
3110 NEXT J
3120 PRINT :"HIT ANY KEY"
3130 CALL KEY(3,KY,ST)
3140 IF ST=0 THEN 3130
3150 I=I+6
3160 IF I>7 THEN 3180
3170 GOTO 2970
3180 RETURN
3190 REM  SAVE CURRENT DATA
3200 X$=""
3210 CALL CLEAR
3220 PRINT "REMOVE PROGRAM C
ASSETTE"
3230 PRINT "& LOAD DATA CASS
ETTE":::
3240 PRINT "HIT ANY KEY":::
::
3250 CALL KEY(3,KY,ST)
3260 IF ST=0 THEN 3250
3270 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
3280 CALL CLEAR
3290 PRINT "STORING BASIC DA
TA":::::::

3300 FOR I=1 TO 12
3310 FOR J=1 TO 3
3320 X$=X$&N$(I,J)
3330 NEXT J
3340 NEXT I
3350 PRINT #1:X$
3360 X$=""
3370 FOR J=1 TO 16 STEP 2
3380 CALL CLEAR
```

```
3390 PRINT "STORING DATA GAM
ES";J;"&";J+1
3400 PRINT :::::::::
3410 FOR I=1 TO 12
3420 X$=X$&A$(I,J)
3430 NEXT I
3440 FOR I=1 TO 12
3450 X$=X$&A$(I,J+1)
3460 NEXT I
3470 PRINT #1:X$
3480 X$=""
3490 NEXT J
3500 CLOSE #1
3510 RETURN
3520 REM  LOAD PREV DATA
3530 CALL CLEAR
3540 X$=""
3550 PRINT "REMOVE PROGRAM C
ASSETTE"
3560 PRINT "& LOAD DATA CASS
ETTE":::
3570 PRINT "HIT ANY KEY"::::
::
3580 CALL KEY(3,KY,ST)
3590 IF ST=0 THEN 3580
3600 CALL CLEAR
3610 OPEN #1:"CS1",INTERNAL,
INPUT ,FIXED 192
3620 CALL CLEAR
3630 PRINT "LOADING BASIC DA
TA"::::::::
3640 INPUT #1:X$
3650 FOR I=1 TO 12
3660 NP=((I*12)+1)-12
3670 N$(I,1)=SEG$(X$,NP,8)
3680 N$(I,2)=SEG$(X$,NP+8,2)
3690 N$(I,3)=SEG$(X$,NP+10,2
)
3700 NEXT I
3710 FOR J=1 TO 16 STEP 2
3720 CALL CLEAR
3730 PRINT "LOADING DATA GAM
ES";J;"&";J+1
3740 PRINT ::::::::
3750 INPUT #1:X$
3760 FOR I=1 TO 12
3770 SP=((I*6)+1)-6
3780 A$(I,J)=SEG$(X$,SP,6)
3790 A$(I,J+1)=SEG$(X$,SP+72
,6)
3800 NEXT I
3810 NEXT J
3820 CLOSE #1
3830 RETURN
3840 REM  VERIFY DATA 1
3850 FOR T1=1 TO LEN(Q$)
3860 Q1=ASC(SEG$(Q$,T1,1))
3870 IF (Q1>47)+(Q1<58)THEN
3900
3880 Q$="X"
3890 T1=L
3900 NEXT T1
3910 RETURN
3920 REM   VERIFY DATA 2
3930 IF LEN(Q$)<>1 THEN 3960
3940 Q1=ASC(Q$)
3950 IF (Q1>47)+(Q1<58)THEN
3970
3960 Q$="X"
3970 RETURN
3980 REM  CHECK AVAIL MEMORY
3990 CALL CLEAR
4000 PRINT "MEM CHECK"
4010 FREMEM=FREMEM+7.9787478
46
4020 GOSUB 4010
```

HAPPY COMPUTING!

CHAPTER EIGHT

Alpha/Numeric Sorting

GENERAL. Now that we've got a good feel for what an array is and how much information you can get into 16K of memory, we can begin the discussion of sorting techniques. There are a number of different methods for sorting a list of numbers, including: Heap Sorts; Bubble Sorts; Tree Sorts; Count Sorts, etc. Voluminous studies, and even complete books, have been written comparing the drawbacks and virtues of each of these methods and their many variations. The fact is "there is no one 'best method' of sorting". There may be one method that is better than another for one particular list, yet another may be superior for a different list. The same old compromises that we have previously discussed apply equally well to sorting. Generally, the less complex sorts are usually the slowest.

In a practical sense, sorting really involves more than just being able to arrange a series of numbers in sequence. It involves the movement and rearrangement of other data based on the results of a sorting process. As an example, let's assume that we have a data file built and loaded into memory that contains the names of twenty of our friends and relatives, as well as their birthdates, anniversary dates, and other important dates. From this file we might want to have the computer print all of the occassions which we need to remember in chronological order. It might be logical to have all of this information stored in a string array where each element contains sufficient room for a family name, the first names of each member, the dates of birth for each member, and an anniversary date. Since each of these data elements contain multiple dates, sorting and printing the list we just mentioned involves more than simply rearranging the 20 elements of the array. What we really need to do is: get all of the individual dates out of each of the 20 elements (maybe three or four from each element); place the dates in an array; and then sort the dates. Now, based on the order in which the dates come out, we need to refer back to our original array (of 20 elements), get the persons name that is associated with that date, and print it to the screen.

Not only are we sometimes working with more than one array, we may also need a sort within a sort. To put it another way, we may have to sort more than one "field" at a time. When we refer to a "field" of data, we're generally talking about a specific segment of a longer line of data. In the above example we had several date fields in each line of data from the array. If we had an array of data that was built chronologically containing, an invoice number, customer account number, and salesman's code, we might want a printout showing each salesman's invoices, arranged in account number order under each salesman. This would require either two individual sorts, or a method of sorting two or more fields simultaneously in one pass through the sort routine. These, as well as other topics, are the subject of this chapter.

Types of Sorts. If you play around very long in the computer world you will hear people making reference to the following commonly used sorting methods:

Bubble Sort	Shell Sort
Selection Sort	Tree Sort
Insertion Sort	Quick Sort

You can also find different versions of these, each claiming to be an improvement on the other. We're going to discuss three of these in some depth -- the Bubble, Insertion & Shell Sorts. In each of the examples below we've used some variable names which hopefully you will not have used in your programs, such as: GROUP(n) as our array to be sorted; FLAG1 to store positions or indicate certain conditions; and HOLD to temporarily remember numbers from the array that need to be arranged. We've put a "beep" into each sort program to indicate where the actual sorting begins and ends, and we've shown the execution time for 5, 20, 50, and 100 numbers in our array. We did not use a RANDOMIZE statement to begin our program because we wanted each program to be run against the same set of numbers. Timing each with a stopwatch and running multiple passes at each level might result in slightly different times than those indicated here; however, these figures do give you a relative comparison of the methods.

Bubble. The bubble sort is probably one of the best known sorting methods. It is also one of the shortest to program and takes the longest amount of time for execution. To explain it, let's look at the following list of 3 numbers:

15 23 17

This program would look at the first two numbers (15 & 23) and compare them. Since the first number is smaller than the second, no change is made in the array and the program goes on to compare the 2nd with the 3rd number (23 & 17). In this case, the computer will exchange the two items and indicate that it's made an exchange by adding 1 to a variable called FLAG1. The new array looks like:

15 17 23

Now, we know that the array is sorted, but the computer doesn't at this point because the value of FLAG1 is not zero. The program resets FLAG1 to Ø and then goes back to item one and compares each of the two adjacent items again. It keeps repeating this process until it can go all the way through the array without causing FLAG1 to increase. At that point the array is sorted. This is the way a true Bubble Sort operates. The following routine is slightly modified and includes a second flag (FLAG2) which is used to indicate where the last exchange was made. This involves only a couple more lines of programming, but it's well worth it for the time saved by not going any "deeper" into the array than is necessary to find any items that are still out of sequence.

```
>1ØØ REM ** MOD BUBBLE **
>11Ø DIM GROUP(1ØØ)
>12Ø N=5
>13Ø CALL CLEAR
>14Ø FOR I=1 TO N
>15Ø GROUP(I)=INT((1ØØØ*RND)+
 1)
>16Ø PRINT GROUP(I);
>17Ø NEXT I
>18Ø CALL SOUND(5Ø,12ØØ,1)
>19Ø PRINT
```

```
>200 FLAG2=N-1
>210 FLAG1=0
>220 ENDP=FLAG2
>230 FOR I=1 TO ENDP
>240 IF GROUP(I)<=GROUP(I+1)T
 HEN 300
>250 FLAG1=FLAG1+1
>260 FLAG2=I
>270 HOLD=GROUP(I+1)
>280 GROUP(I+1)=GROUP(I)
>290 GROUP(I)=HOLD
>300 NEXT I
>310 IF FLAG1>0 THEN 210
>320 CALL SOUND(50,1200,1)
>330 PRINT
>340 FOR I=1 TO N
>350 PRINT GROUP(I);
>360 NEXT I
>370 GOTO 370
>RUN
```

Results: 2 Seconds with N=5
7 Seconds with N=20
37 Seconds with N=50
162 Seconds with N=100

Try this yourself and change some of the values. To really see what's happening, add the following two lines to your program and run it again.

```
>225 PRINT
>235 PRINT "COMP ";GROUP(I);"
 &";GROUP(I+1);"FLAG=";FLAG1
```

While this type of sort exchanges more than one number in a single pass, it can be very time consuming if you have a long list with a low number near the very end. Numbers will "bubble" up the list to the top only one spot per pass through the array even if all of the other numbers are in order. Considering that some of the other methods are almost as short, and much faster, we feel this method has limited usefulness.

The Insertion Sort. The insertion sort, in addition to the fact that it is quicker and consumes the same number of lines as the bubble sort, is also easier to understand. This sort starts with item one in the array and goes all the way through, looking for and saving the value and position of the lowest number (the value is saved as HOLD and the position as FLAG1 in the program below). When this is done, it moves all of the items above that position down one position, thus leaving position one empty. Position 1 is now replaced with the lowest number. That's the first pass. On the second and subsequent passes it starts its search one more down the array (position 2, then 3, etc). It's not necessary to check the last one, since it's obviously the lowest remaining if there's only one. The program below, with the print modifications we'll give you at the end, should give you an adequate explanation of the process.

```
>100 REM *INSERTION SORT *
>110 DIM GROUP(100)
>120 N=5
>130 CALL CLEAR
>140 FOR I=1 TO N
>150 GROUP(I)=INT((1000*RND)+
 1)
>160 PRINT GROUP(I);
>170 NEXT I
>180 CALL SOUND(50,1200,1)
>190 PRINT
>200 FOR J=1 TO N-1
>210 HOLD=1001
>220 FOR I=J TO N
>230 IF GROUP(I)>HOLD THEN 26
 0
>240 HOLD=GROUP(I)
>250 FLAG1=I
>260 NEXT I
>270 FOR K=FLAG1 TO J+1 STEP
 -1
>280 GROUP(K)=GROUP(K-1)
```

```
>290 NEXT K
>300 GROUP(J)=HOLD
>310 NEXT J
>320 CALL SOUND(50,1200,1)
>330 PRINT
>340 FOR I=1 TO N
>350 PRINT GROUP(I);
>360 NEXT I
>370 GOTO 370
>RUN
```

Results: 1 Seconds with N=5
5 Seconds with N=20
26 Seconds with N=50
96 Seconds with N=100

Notice that, while both of the above line listings really only run from 200 to 310, the times are significantly improved. By making the following changes you can observe what happens to the array on each pass through.

```
>190 PRINT ::
>305 PRINT "PASS=";J;"HOLD#="
 ;HOLD;"POS=";FLAG1
>306 GOSUB 340
>325 GOSUB 340
>326 GOTO 326
>365 PRINT ::
>370 RETURN
```

<u>Shell Sort.</u> For our purposes, we recommend almost universal use of the following program. The operational portion consumes only 2 more lines than the above programs (it runs from 200-330), yet it sorts 100 numbers in about 1/3 the time required by the next best sort. The program makes successive passes through the array, performing comparisons as it goes, except that it does not compare adjacent items as does the bubble sort. Instead, it compares items separated by an interval defined below as the DIF (for difference). After each complete pass through the main loop, which begins in line 220, the difference is cut in half and the process starts over. This continues until DIF=0, at which time the sort is completed. Within the main FOR-NEXT statement, comparisons are made, exchanges are made when required, and FLAGs are reset. It's success lies in the fact that this system requires far fewer comparisons than the previous methods.

```
>100 REM * SHELL SORT *
>110 DIM GROUP(100)
>120 N=5
>130 CALL CLEAR
>140 FOR I=1 TO N
>150 GROUP(I)=INT((100*RND)+1
 )
>160 PRINT GROUP(I);
>170 NEXT I
>180 CALL SOUND(50,1200,1)
>190 PRINT ::
>200 DIF=INT((N*1.5)/2)
>210 IF DIF=0 THEN 340
>220 FOR I=1 TO N-DIF
>230 FLAG1=I
>240 FLAG2=FLAG1+DIF
>250 IF GROUP(FLAG1)<=GROUP(F
 LAG2)THEN 310
>260 HOLD=GROUP(FLAG1)
>270 GROUP(FLAG1)=GROUP(FLAG2
 )
>280 GROUP(FLAG2)=HOLD
>290 FLAG1=FLAG1-DIF
>300 IF FLAG1>0 THEN 240
>310 NEXT I
>320 DIF=INT(.5*DIF)
>330 GOTO 210
>340 CALL SOUND(50,1200,1)
>350 GOSUB 370
>360 GOTO 360
>370 FOR R=1 TO N
>380 PRINT GROUP(R);
>390 NEXT R
>400 RETURN
>RUN
```

Results: 1 Seconds with N=5
3 Seconds with N=20
12 Seconds with N=50
29 Seconds with N=100

You can gain a better understanding of how this operates by adding the following lines to the program:

```
>120 N=6
>245 PRINT "PS1=";FLAG1;"PS2=
 ";FLAG2;"DIF=";DIF;"I=";I
>295 GOSUB 370
>296 PRINT ::
```

This display starts by showing the original array (of 6 numbers). It then shows you which item numbers it is comparing, the value of DIF, and the value of I, as the sort takes place. After each exchange, it shows you the new array. In all of our programs involving sorts, we use this method.

Tree (Heap) Sort. A Tree Sort requires approximately twice as many lines of code as the Shell Sort and takes just slightly longer to complete most sorts.

Selection Short. As far as time of completion, the Selection Sort is about equal to the Insertion Sort. It can be written with about 2 less lines of code than either the bubble or insertion sort. It works with a pair of nested loops which start with the first item in the array and then, using the inner loop, comparisons are made between this item and all others (just like a bubble sort). This process continues, using the second item, third, etc., until all numbers have been compared with all others.

Whereas some of the other methods will sort a "nearly" sorted list faster than one that is highly randomized, this program is always consistent in terms of the number of passes required.

Quick Sort. A Quick Sort requires about three times as many lines of code; however, it will operate slightly faster than the Shell Sort. In quantities of approximately 100 items the actual time difference will amount to no more than 4 or 5 seconds. The best way to describe this is to compare it to putting a shuffled deck of cards back in order. You might start by going through the deck and putting all red cards in one stack and all black cards in another stack. Next, you could take the red deck and divide it into stacks of diamonds and hearts. Now, take the stack of diamonds and divide it approximately in half, with numbers below 7 in one stack and over seven in another. You could keep subdividing the diamond stack until finished and then start doing the same thing with each of the other stacks. The quick sort operates essentially on this principle. It actually places the last number in the array in the middle and then starts subdividing each half, over and over, until completed.

Unless you want to pursue the matter of sorting as a science in its own right, you are probably going to be better off to pick one, learn it, and use it whenever a sort is required. Deviate from it only when you have a unique situation for which one may be better than another, or where speed is absolutely essential. Our vote goes to the Shell Sort as the best compromise of ease of programming and speed.

Order Preference. In the above examples we were sorting from low to high. For golf scores or dates, this might be fine, but what about test scores, bowling scores, sales figures? For these we may want to go from highest value to lowest. In order to do this, simply change the operators in line 250 from "<=" to ">=". This is the only change required to reverse the order.

Alphabetic Sorts. Let's not make this task any more difficult than it is. Looking at the Shell Sort, the only thing required is to add a $ (dollar sign) after every reference to GROUP. Our array is then called GROUP$(n). You'll also have to put a $ after the HOLD variable in lines 260 and 280. The only caution here is to remember that in order to sort alphabetically you must have an adequate array to work from. This means that the "sort field" must be identifiable and all characters in the sort field should be upper case. As an example, suppose you were building a data file containing both first and last names, and that you were going to want to sort it by last name. If your input line simply asked for first & last name and then assigned that directly to the array, you might wind up with an array containing entries as follows: Johnny Johnson, Hank Williams, Bill Johns, Tom Kennedy, and Howard Yancy.

Using the shell sort program as a base, make the following modifications to see how this works.

First, add $ after all GROUP references and the word HOLD, and remove the semicolon from the PRINT statement in 160 and 380. Now add the following lines:

```
>135 DATA Johnny Johnson,Hank
  Williams,Bill Johns,Tom Ken
 nedy,Howard Yancy
>150 READ GROUP$(I)
```

Running this through the Shell Sort returns these in the following order: Bill Johns, Hank Williams, Howard Yancy, Johnny Johnson, and Tom Kennedy. That's OK if you want it by first name, but for most purposes it isn't what you would want. If we turn our original list around and put last name first and first name second, the results are correct and as follows: Johns Bill, Johnson Johnny, Kennedy Tom, Williams Hank, and Yancy Howard. In the above example, we've used upper and lower case and the results were correct. Replace the data line (135) with the following and run the sort routine again:

```
>135 DATA MacWilliams,MacKnig
 ht,Mackey,Macomber,Maccione
```

This routine places MacKnight and MacWilliams ahead of all others which are listed strictly in lower case. In alphabetical order, we should have MacKnight following Mackey and MacWilliams at the end of the list, regardless of capital letters. The only way to insure a proper sort is to use all upper case letters. Further, if you put a space between the Mac and Williams, it will appear first in the list.

Looking at the screen display, think of each letter from left to right as a row of characters. "M's" are in the first row, and "A's" in the second in the above example. What the computer actually does is evaluate the ASC character value of each letter in a particular row and the lower values are placed first. A space (value of

32) will always come before any letter and all upper case letters will come before lower case letters. This is because even a capital "Z" has a lower value than a lower case "A". In most instances, building your data file all in upper case should present no problem, the exception might be in word processing, where you want a nice letter made out to "John McWilliams", not "JOHN MCWILLIAMS". In this case you might actually need two fields in the data file -- one with the name for sorting purposes and another with the name for printing (or display) purposes.

Sort Fields. Let's approach this sorting logically. First, if you only have four or five items you probably aren't going to sort them at all. These items you would just display on the screen and you could mentally figure it out. If you have a lot of items, normally you won't be generating this from screen input, but from a data file. If you run it in from a data file you normally will be putting it into one big array. We're going to use the "Memory Jogger" program as an example and show you how this process takes place and how the sorting arrays are built.

In the "Jogger" program we have each family unit set up as a 63 character string. Each string holds: one last name; up to three first names; three birthdates; and an anniversary date. The "field" where each is located is as follows:

Last Name - Position 1-15
First Name - 16-23 Birthday - 24-29
First Name - 3Ø-37 Birthday - 38-43
First Name - 44-51 Birthday - 52-57
Anniversary - 58-63

There are three of these 63 character groupings on one data file line, 189 characters long. This is the way it's printed to the cassette and retrieved (inputted) from the cassette. When we INPUT from the data file we break this back down into family units (63 characters) and store each one of these strings as FMY$(n). We also keep track of how many family units there are using the variable CNT1.

Now, every time we use the menu to request either an ALPHA sort or DATE sort, the first thing we do is run the program through a special subroutine at line 253Ø that builds two special sorting arrays called NM$ (for all names, both last and first) and DT$ (for all dates, both birthday and anniversary). Since there can be as many as four names and four dates in a single family unit, these sorting arrays are four times as long as the CNT1 value. Analyze this subroutine and you'll notice how we "strip" each one of these items out of the FMY$(n) array. After each one of the names that we store, we also add a variable called I$, which represents the value FMY$(n). For instance, if FMY$(12) had a first name of "JOHN" in it, it would be stored in the array NM$ as "JOHN 12". Each first name field is 8 characters long and it has the number 12 following that. The last name would be 15 characters long, followed by a two digit number.

For the dates, we need still further information. To the date, we add three other bits of information. First, we add a character, either "A" or "B" to indicate whether the date is a birthday or anniversary. Second, we add the I$ value explained above. Third, if it's a birthday, we indicate whether it's the birthday of the first, second, or third name in the

family unit. For instance, if the third name in the FMY$(11) for JOHNSON was "CHAD" with a birthdate of 112478, the DT$ would be: "112478B113". These two arrays (NM$ and DT$) are the arrays that will be sorted, not the FMY$.

Both of these arrays can be sorted using essentially the same Shell Sort routine previously shown. You can find this in lines 1230-1530 of the "Jogger" program. We've modified this just slightly by checking for a variable "Q" in line 1340. If Q=4 then we use the array DT$ in our comparisons, otherwise we use the array NM$. After the sort is completed we go to the display portion of the program.

To print the information to the screen in date order it's very easy to just read down through the DT$ array. Since the computer sorts from left to right, and the left most portion of the array is the month, followed by the day of the month, and then year, etc., this display will provide us with all of January's occasions, then February's, etc. As we come to each item in the array we can determine immediately if it's a birthday or anniversary, based on the code. Then, we look at the value for I$ and get the last name from the FMY$(n) array. Now, based on the value of the last number in our DT$ array, we can calculate where, in FMY$, we'll find the first name associated with a birthdate. The approach for NM$ is not as complicated, since all we need is the code at the end of the NM$ to determine which element of FMY$ we want.

Sorting Dates. The way the date is stored in a sort array is important. The above example used the date just as you would normally enter it, with month, day, and year, in that order. We weren't concerned with the actual chronological order in this case, just the sequence of events within any given year. However, if you were dealing with sales, test scores, bowling scores, or many other types of data, you might want the true order of progression. In this case, either when you store the information in your original array or when you build your sort file, you'll have to change the way the date is listed. If we had a date variable called DAT$ that was equal to "Ø72376", we might use a simple command like the following to put it in a different order:

```
DAT$=SEG$(DAT$,5,2)&SEG$(DAT$(1,4)
```

This idea of putting the year in front of the rest of the information will also be discussed later in this manual, since it's very important when you try to determine intervals of time between two dates: for instance, to determine all sales calls made with the last 45 days, accounts receivables 3Ø days overdue, etc.

Sorting Multiple Fields. Suppose we had a data file which held the date of quarterly tests, the score of each test, and the names of 25 students. These probably would have been recorded in date order, but the scores and names may have been random. A few entries might look like:

```
Ø32883JOHNSON     82.5
Ø32883ADAMS       89.2
Ø32883WILSON      75.6
```

With this type of situation, it's very possible that we might want to be able to display each test period, with each student listed alphebetically followed by his test score. Running these data

lines directly through our Shell Sort will result in the proper sequence. If we wanted them in order by test score within each scoring period we would have to rebuild the data line to as follows:

```
Ø3288382.5JOHNSON
Ø3288389.2ADAMS
Ø3288375.6WILSON
```

Your print routine (either to screen or printer) may remain the same; however, your sort array would have to be constructed as shown. We are actually performing three sorts in the above example. If you wanted all students alphabetically with each of their quarterly test scores you would simply list name first, followed by date, and then score. There is no need to go through the sort routine more than once, just build your array accordingly. In most cases you'll only need to key in one version of a sort into your program. With a couple of "IF" statements added to the sort, the way we used the "Q" variable above, you can reconstruct the data file just prior to where it makes its comparison.

<u>Disk Drive Applications.</u> We seldom get into a discussion of disk drive or other expansions; however, in this case a brief mention does seem appropriate. When dealing with console basic, we are always bringing all data into memory prior to sorting (such as the FMY$ above). In some cases, it might be possible to sort this entire array directly, instead of creating additional arrays on the side for sorting purposes. If there's no particular necessity for it being stored in a specific order, this presents no problem. If you ever do expand to disk drive you'll find that this is not always convenient or wise. On disk drive you may have 3ØØ data lines, each containing 15Ø characters of information. This may include names, addresses, telephone numbers, dates, scores, etc. If they are account names, each may have an account number and you want them permanently stored that way. It's impossible, in 16K of memory, to pull all of this information in and sort it at one time. Using the above method, we keep only the necessary sorting information in memory, perhaps 5 or 6 characters out of each data line of 15Ø characters. We also carry with that information the "RECORD" number. If you know the record number, using relative files, it's very simple to retrieve any other information from the disk for later processing or printing to the screen or printer.

```
*****************
*  MEMORY JOGGER *
*    V-PO831KB   *
*   BY T CASTLE  *
*****************
```

DESCRIPTION. Did you ever remember at the last minute that you didn't get a card or present for someone you knew who had a birthday or anniversary? Are you in charge of a directory committee or welcoming committee for a subdivision? With this little program you can keep track of all of those important dates.

The program permits you to build a data file on up to thirty families. For each family you're asked to input the family name, up to three first names, a birthday for each, and an anniversary date. Using the display menu, you can call these back up in alphabetical order (by last name) and it'll display all information for each family. The screen shows four families at a time and then asks you to "HIT ANY KEY". It keeps progressing through the list until all active families are displayed. A second option permits a display of all occasions in date order, by month and day of the month. About the 2Øth or 25th of each month you can sit down and list all of the events for the coming month. Each name has a key to the left indicating whether it is a birthday (B) or anniversary (A). The screen instructions are straight forward and self explanatory. The program works perfect for a family of three (Mom, Dad, and the Little One). A single person or family of two can be entered. Simply keep hitting the "ENTER" key to bypass all other questions. For families over three, reenter the last name and additional family members. This program is prime for modification to serve other purposes. A salesman could use it to keep track of new prospects by changing the "LAST NAME" to "COMPANY NAME". He could then use the spot designated for first names as "FOLLOW UP 1", "FOLLOW UP 2", and "FOLLOW UP 3". Even without changing the names, you can use it to record any recurring event such as quarterly taxes, fertilizing the lawn, and oil changes on the auto. Nobody said it really had to be a "FAMILY NAME", just call it an "EVENT".

NOTES. Since it involves mostly sorting, the construction of this program, including the descriptions of the arrays and data fields, is covered fully in the chapter on Sorting. The input and output sections are in our normal format, using full length 192 character data lines. Each one contains three family groups. At full capacity, ten read statements are required from the cassette recorder. It takes a little over a minute to load. The most time consuming portion of the program is the sort. The program sorts immediately after old data is loaded and after any new entry or change. Several minutes are required since it must strip the valid information from the FMY$(n) array and sort two arrays with up to 12Ø elements each. There isn't much that can be done to speed the first sort after the previous data is loaded; however, if you're ambitious and the time lag bothers you, you could write an additional subroutine that would wedge the new names into each of the three presorted arrays (FMY$, NM$, & DT$). At full capacity remaining memory is about 18ØØ bytes.

```
100 REM   *****************
110 REM   * MEMORY JOGGER *
120 REM   *****************
130 REM
140 REM   BY T CASTLE
150 REM   AMLIST V-P0831KB
160 REM
170 REM   INITIAL VARIABLES
180 CALL CLEAR
190 DIM FMY$(30),DT$(120),NM
$(120)
200 AD$="               "
210 REM   MAIN MENU
220 CALL CLEAR
230 PRINT TAB(10);"MAIN MENU
"::
240 PRINT "  1. INPUT PREVIO
US DATA"::
250 PRINT "  2. INPUT/CHANGE
 INFO"::
260 PRINT "  3. ALPHA DISPLA
Y"
270 PRINT "     FAMILY BY LA
ST NAME"::
280 PRINT "  4. DATA DISPLAY
"
290 PRINT "     ALL OCCASION
S"::
300 PRINT "  5. SAVE DATA"::
310 PRINT "  6. EXIT PROGRAM
"::::
320 INPUT "SELECTION? ":Q
330 IF (Q<1)+(Q>6)THEN 320
340 IF Q=6 THEN 370
350 ON Q GOSUB 2010,390,1550
,1750,2270
360 GOTO 220
370 STOP
380 REM   INPUT/CHANGE DATA
390 CALL CLEAR
400 SRT=0
410 PRINT "ARE YOU CHANGING
EXISTING   DATA OR   ADDING
NEW DATA"
420 INPUT "ENTER (C OR N)? "
:Q$
430 IF Q$="N" THEN 760
440 IF Q$<>"C" THEN 420
450 CALL CLEAR
460 PRINT "ENTER LAST NAME O
F FAMILY"
470 INPUT "TO CHANGE: ":Q$
480 Q$=SEG$((Q$&AD$),1,15)
490 CALL CLEAR
500 FOR I=1 TO CNT1
510 IF Q$<>SEG$(FMY$(I),1,15
)THEN 740
520 PRINT ::::SEG$(FMY$(I),1
,15)&"  ANV-"&SEG$(FMY$(I),5
8,6)
530 PRINT " "&SEG$(FMY$(I),1
6,8)&" "&SEG$(FMY$(I),24,6)
540 PRINT " "&SEG$(FMY$(I),3
0,8)&" "&SEG$(FMY$(I),38,6)
550 PRINT " "&SEG$(FMY$(I),4
4,8)&" "&SEG$(FMY$(I),52,6):
:::
560 PRINT "IF ANY ITEM ABOVE
  REQUIRES A CHANGE, ENTIRE
ENTRY MUST"
570 PRINT "BE DELETED & REEN
TERED."::
580 PRINT "MAKE NOTE OF CORR
ECT INFORM-ATION  BEFORE  DE
LETING  AND"
590 PRINT "USE NEW OPTION TO
 REENTER"::
600 PRINT "USE 'S' TO BYPASS
"::::
610 PRINT "ENTER (D) TO DELE
TE OR"
620 INPUT "ENTER (S) TO SEAR
CH? ":Q$
630 IF Q$="S" THEN 740
640 FOR J=I TO CNT1-1
650 FMY$(J)=FMY$(J+1)
660 NEXT J
670 FMY$(CNT1)=""
680 CNT1=CNT1-1
690 FOR K=1 TO 120
700 NM$(K)=""
710 DT$(K)=""
720 NEXT K
730 I=CNT1
740 NEXT I
750 GOTO 1210
760 CALL CLEAR
770 IF CNT1=30 THEN 1210
```

```
780 PRINT "ENTER  LAST  NAME
 OF FAMILY,1ST NAME OF UP TO
 THREE FAM-ILY MEMBERS, W/DA
TE OF BIRTHFOR EACH."
790 PRINT "ENTER  ANNIVERSAR
Y DATE  FORFAMILY.  IF QUEST
ION DOESN'TAPPLY, HIT ENTER
KEY"::::
800 INPUT "LAST NAME  ":EN1$
810 IF LEN(EN1$)>15 THEN 800
820 EN1$=SEG$((EN1$&AD$),1,1
5)
830 PRINT
840 INPUT "1ST PERSON ":EN2$
850 IF LEN(EN2$)>8 THEN 840
860 EN2$=SEG$((EN2$&AD$),1,8
)
870 INPUT "BIRTH DATE ":Q$
880 GOSUB 2680
890 IF Q$="X" THEN 870
900 EN3$=Q$
910 PRINT
920 INPUT "2ND PERSON ":EN4$
930 IF LEN(EN4$)>8 THEN 920
940 EN4$=SEG$((EN4$&AD$),1,8
)
950 INPUT "BIRTH DATE ":Q$
960 GOSUB 2680
970 IF Q$="X" THEN 950
980 EN5$=Q$
990 PRINT
1000 INPUT "3RD PERSON ":EN6
$
1010 IF LEN(EN6$)>8 THEN 100
0
1020 EN6$=SEG$((EN6$&AD$),1,
8)
1030 INPUT "BIRTH DATE ":Q$
1040 GOSUB 2680
1050 IF Q$="X" THEN 1030
1060 EN7$=Q$
1070 PRINT
1080 INPUT "ANVSR DATE ":Q$
1090 GOSUB 2680
1100 IF Q$="X" THEN 1080
1110 EN8$=Q$
1120 PRINT ::"TO VERIFY, ENT
ER (V) OR TO"
1130 INPUT "REENTER, ENTER (
R)?       ":Q$
1140 IF Q$="V" THEN 1190
1150 IF Q$="R" THEN 1160 ELS
E 1120
1160 CALL CLEAR
1170 PRINT "INFORMATION REJE
CTED-REENTER"::::::
1180 GOTO 800
1190 CNT1=CNT1+1
1200 FMY$(CNT1)=EN1$&EN2$&EN
3$&EN4$&EN5$&EN6$&EN7$&EN8$
1210 RETURN
1220 REM   ALPHA/DATE SORT
1230 CALL CLEAR
1240 IF SRT=1 THEN 1530
1250 PRINT "REBUILDING & SOR
TING. . ."
1260 Q=4
1270 GOSUB 2530
1280 N=CNT1*4
1290 DIF=INT((N*1.5)/2)
1300 IF DIF=0 THEN 1490
1310 FOR I=1 TO N-DIF
1320 FLAG1=I
1330 FLAG2=FLAG1+DIF
1340 IF Q=4 THEN 1400
1350 IF NM$(FLAG1)<=NM$(FLAG
2)THEN 1460
1360 HOLD$=NM$(FLAG1)
1370 NM$(FLAG1)=NM$(FLAG2)
1380 NM$(FLAG2)=HOLD$
1390 GOTO 1440
1400 IF DT$(FLAG1)<=DT$(FLAG
2)THEN 1460
1410 HOLD$=DT$(FLAG1)
1420 DT$(FLAG1)=DT$(FLAG2)
1430 DT$(FLAG2)=HOLD$
1440 FLAG1=FLAG1-DIF
1450 IF FLAG1>0 THEN 1330
1460 NEXT I
1470 DIF=INT(.5*DIF)
1480 GOTO 1300
1490 IF Q=3 THEN 1520
1500 Q=3
1510 GOTO 1280
1520 SRT=1
1530 RETURN
1540 REM  ALPHA DISPLAY
```

```
1550 GOSUB 1230
1560 K=0
1570 FOR I=1 TO (CNT1*4)+1
1580 IF I=(CNT1*4)+1 THEN 16
70
1590 IF LEN(NM$(I))<>17 THEN
 1720
1600 J=VAL(SEG$(NM$(I),16,2)
)
1610 K=K+1
1620 PRINT SEG$(FMY$(J),1,15
)&"  ANV-"&SEG$(FMY$(J),58,6
)
1630 PRINT " "&SEG$(FMY$(J),
16,8)&" "&SEG$(FMY$(J),24,6)
1640 PRINT " "&SEG$(FMY$(J),
30,8)&" "&SEG$(FMY$(J),38,6)
1650 PRINT " "&SEG$(FMY$(J),
44,8)&" "&SEG$(FMY$(J),52,6)
::
1660 IF K<4 THEN 1720
1670 PRINT "HIT ANY KEY"
1680 CALL KEY(3,KY,ST)
1690 IF ST=0 THEN 1680
1700 K=0
1710 CALL CLEAR
1720 NEXT I
1730 RETURN
1740 REM  DATE DISPLAY
1750 GOSUB 1230
1760 K=0
1770 FOR I=1 TO (CNT1*4)+1
1780 IF I=(CNT1*4)+1 THEN 19
30
1790 IF LEN(DT$(I))<>10 THEN
 1980
1800 IF SEG$(DT$(I),1,6)="00
0000" THEN 1980
1810 J=VAL(SEG$(DT$(I),8,2))
1820 MK=((VAL(SEG$(DT$(I),10
,1)))*14)+2
1830 K=K+1
1840 PRINT SEG$(DT$(I),7,1);
" ";
1850 PRINT SEG$(DT$(I),1,6);
" ";
1860 PRINT SEG$(FMY$(J),1,10
);
1870 IF SEG$(DT$(I),7,1)<>"B
" THEN 1900
1880 PRINT SEG$(FMY$(J),MK,8
)
1890 GOTO 1910
1900 PRINT
1910 PRINT
1920 IF K<10 THEN 1980
1930 PRINT "HIT ANY KEY"
1940 CALL KEY(3,KY,ST)
1950 IF ST=0 THEN 1940
1960 K=0
1970 CALL CLEAR
1980 NEXT I
1990 RETURN
2000 REM  LOADS PREV DATA
2010 CALL CLEAR
2020 ECK=0
2030 PRINT "REMOVE PROGRAM C
ASSETTE AND LOAD DATA CASSET
TE"::
2040 PRINT "HIT ANY KEY"::::
::
2050 CALL KEY(3,KY,ST)
2060 IF ST=0 THEN 2050
2070 OPEN #1:"CS1",INTERNAL,
INPUT ,FIXED 192
2080 CALL CLEAR
2090 PRINT "LOADING DATA":::
:::::
2100 DF=1
2110 CNT1=0
2120 INPUT #1:X$
2130 FOR I=DF TO DF+2
2140 TMP$=SEG$(X$,(I*63)-((D
F*63)-1),63)
2150 IF SEG$(TMP$,1,15)="
               " THEN 2190
2160 FMY$(I)=TMP$
2170 TMP$=""
2180 CNT1=CNT1+1
2190 NEXT I
2200 DF=DF+3
2210 IF SEG$(X$,190,1)<>"X"
THEN 2120
2220 CLOSE #1
2230 CALL CLEAR
2240 GOSUB 1230
2250 RETURN
2260 REM  SAVES DATA
2270 CALL CLEAR
```

```
2280 PRINT "REMOVE PROGRAM C
ASSETTE AND LOAD DATA CASSET
TE"::
2290 PRINT "HIT ANY KEY"::::
::
2300 CALL KEY(3,KY,ST)
2310 IF ST=0 THEN 2300
2320 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
2330 CALL CLEAR
2340 PRINT "STORING DATA":::
:::::
2350 X$=""
2360 J=0
2370 FOR I=1 TO CNT1
2380 J=J+1
2390 X$=X$&FMY$(I)
2400 IF I=CNT1 THEN 2420
2410 IF J=3 THEN 2460 ELSE 2
490
2420 IF LEN(X$)=189 THEN 245
0
2430 X$=X$&" "
2440 GOTO 2420
2450 X$=X$&"X"
2460 PRINT #1:X$
2470 J=0
2480 X$=""
2490 NEXT I
2500 CLOSE #1
2510 RETURN
2520 REM  BLDS SORT ARRAY
2530 FOR I=1 TO CNT1
2540 J=(I*4)-3
2550 I$=SEG$((STR$(I)&AD$),1
,2)
2560 NM$(J)=SEG$(FMY$(I),1,1
5)&I$
2570 DT$(J)=SEG$(FMY$(I),58,
6)&"A"&I$&"0"
2580 L=0
2590 FOR K=16 TO 44 STEP 14
2600 L=L+1
2610 MK$=STR$(L)
2620 NM$(J+L)=SEG$(FMY$(I),K
,8)&I$
2630 DT$(J+L)=SEG$(FMY$(I),K
+8,6)&"B"&I$&MK$
2640 NEXT K
2650 NEXT I
2660 RETURN
2670 REM  VERIFY DATA
2680 IF LEN(Q$)=6 THEN 2720
2690 IF LEN(Q$)>0 THEN 2790
2700 Q$="000000"
2710 GOTO 2800
2720 FOR I=1 TO 6
2730 J=ASC(SEG$(Q$,I,1))
2740 IF (J<48)+(J>57)THEN 27
50 ELSE 2760
2750 Q$="X"
2760 NEXT I
2770 IF Q$="X" THEN 2800
2780 GOTO 2800
2790 Q$="X"
2800 RETURN
```

HAPPY COMPUTING!

CHAPTER NINE

Validity & Testing

GENERAL. Nothing is more annoying than spending 30 or 40 minutes inputting data, only to have the program "error out" or "bomb" in a non-recoverable position (which means you've wasted your time). If this happens while you're inputting data, more than likely you're a victim of the old data processing adage "Garbage In - Garbage Out". The program may have been looking for numeric data and you entered string data or a string value was too long. Even if it's just a game you're playing, it seems that the most likely time for it to error out will be when you're 100 points short of a record. In this case the computer probably generated some variables that were out of range for screen display; frequency or volumes out of range; character codes, etc.

A well written program contains sufficient validation and testing routines to prevent these kinds of errors. Usually, testing is done with the IF - THEN or IF - THEN - ELSE statements. You've seen these statements numerous times throughout our programs. The interesting thing is that probably 30-50% of these could be eliminated, and the program would function properly, provided that the operator never hit a key he wasn't supposed to. Knowing when, where and how to use these validation lines would seem relatively easy, yet many supposedly complete programs still "bomb". During the initial development of a program many of the validity statements may be bypassed to save time until a fully functional program is created that completes its cycle from beginning to end. Then, depending on who's going to use it and how permanent it is, additional statements can be added at significant points to prevent malfunctions. If a program still errors out on occasion, in most cases it's not because the programmer didn't know how to prevent the error, he just didn't consider the possibility that a particular situation might occur. What we're going to do in this chapter is point out some of the obvious, and less obvious, places where testing is required, and to give you some methods and precautions you can take to prevent errors.

Keyboard Input. Let's face it, between the operator and the computer, the most likely one to make a mistake is the operator, so keyboard input is the most obvious place to start our discussion. There are four standard questions that you can ask yourself for every INPUT statement you have in the program.

1. Have I made it clear, someplace prior to the input statement, what kind of information I'm looking for?
2. What can I do after input to check the technical specifications of the data (length, string, numeric, etc)?
3. What action needs to be taken if it isn't correct?
4. Even if the input is technically valid, should it or can it be tested to see if it is logical?

Action Before Input. Once you hit an INPUT line, there's no turning back. In order for the program to continue the user must do something, if only hit the enter key. How much you say prior to input depends on the nature of the program and exactly what's going to happen with the input. If this program is a "one time" utility program that is used only by you, the leading information can be very brief or completely omitted. In a program like the "Money Planner", which asks for amounts, interest rates, etc., we ask for numeric information using a numeric variable, and we don't bother to show you, in the program, how to enter interest rates (1Ø% or .1Ø); however, we do cover it in the documentation. The reason is that no data is going to be permanently modified or stored if you make a mistake, nor is much time going to be wasted. If it does error out, you simply run it again and there's no harm done. If you have a lot of data to input and an experienced operator is going to be using the program you can go to something often referred to as "Speed Input". In one section of the bowling program we ask for series, roster number, and scores for 3 games, using five input statements. If you're the only one that ever enters data and you don't care about a nice screen display, you could simply use a statement like:

```
>1ØØ INPUT "SER,R#,G1,G2,G3?":
 WK,I,G1,G2,G3
```

As long as this information is verified after input, and prior to modifying the permanent data file arrays, it might be fine for an adult; however, in something like the "Baseball Stats" program for young baseball players (who may actually want to enter the data), the chance of an error is greatly increased. If the program is for multiple users, young people, or it involves permanent data files, you're wise to use separate input statements rather than combine them into one. Further, prior to each statement, use a couple of print lines to explain, and preferably give them an example, of exactly what you want. For instance, tell them you want: dollar amounts, with no decimals (i.e. 12ØØ not $1,2ØØ.ØØ); dates (i.e. Ø81683); or an answer of Yes or No (Y or N). The more programming you do, the easier it will be for you to take for granted some things that aren't so obvious to someone less experienced. Now let's discuss the variables to use for inputting data.

If you're inputting information that will go into an array or data file, it's prudent never to use the actual active variable (such as FMY$(1Ø)), until the information has been thoroughly validated. As a standard practice, we'll use something called "Q" or "Q$", as a temporary "holding" variable, that we can either accept or reject, without ever altering our permanent and operable data. In many of our programs, even if we really want numeric input, we ask for string data. We often use a variable simply called Q$. If you ask for a value (like "A") and someone puts in a number with an "O" instead of a zero (Ø), the program will not error out; however, you will be greeted with a loud buzz and the warning message:

```
* WARNING:
  INPUT ERROR IN (line number)
TRY AGAIN:
```

Again, this may be alright for the experienced user, but it could be rather threatening to the novice. In addition, it causes the screen to

scroll up four lines every time you make an error. If your input requires that they refer to other information already on the screen, this could cause a problem. To be safe, input all data as string data and let the computer analyze it and decide what action to take.

There are a couple of other places where specific warnings are recommended prior to going to the INPUT statement. One of these is when you've asked them for a record number or name that is going to be deleted from a file. If they hit the wrong key on the menu selection and wind up in a "delete" portion by accident, you should give them a warning of what's going to happen and a way to avoid it if they need to. Second, before permitting them to exit a program which creates data files, you should pose the question, "HAVE YOU SAVED YOUR DATA?", or some other similar type warning.

Action After String Input. Usually, string data winds up going into an array or being displayed on the screen. If it does, it's usually necessary to have it be a specific length and either right justified or left justified. After each input of a string variable you should ask yourself the following five questions:

1. What if the string is too long?
2. What if it's too short (or nothing)?
3. What if it's just right?
4. If it's the right length, is it valid?
5. Can I fix it or not, and if not, what should I do?

The answer to each of these questions isn't important, provided that the solution to each keeps you within the program and that it doesn't cause or allow the computer to error out.

```
>100 INPUT "NAME? ":Q$
>110 IF LEN(Q$)>8 THEN 150
>120 IF LEN(Q$)=8 THEN 170
>130 Q$=" "&Q$
>140 GOTO 120
>150 PRINT "TOO LONG"
>160 GOTO 100
>170 REM LENGTH OK, ACCEPT OR
 TEST FURTHER
>180 NAME$=Q$
```

The above routine adequately answers, or at least considers, each of the possibilities. Your choice of line length may be different; you may decide to send it directly back to the input line if it's too long, instead of an error message; you may want it left justified instead of right justified, so you would put the space after the Q$ in line 130; or you may want to compare the name with names in an existing array in line 170. If you simply hit the ENTER key with the above routine, it would create a string 8 characters long filled with spaces. You may want to force an input and add an extra line that causes a zero length input to repeat the question or give an error message.

Action After Numeric Input. Going on the assumption that you're going to use our suggestion and input some numerical information as string data, after each such input you should ask the following:

1. Do all the characters in this string represent numeric values?
2. Do I want decimals?
3. If I have decimals, how many places do I want following?

4. Will I need to round the number up or down?
5. Do I want it converted back to a string?
6. If I do, should I right or left justify?

As above, the answers to these questions will vary from program to program; but, they should all be answered one way or another in the statements following the input line. If the number is entered as Q$, you can test it as a number by running it through the following routine:

```
>100 INPUT "NUMBER:  ":Q$
>110 IF LEN(Q$)=0 THEN 100
>120 FOR I=1 TO LEN(Q$)
>130 IF (ASC(SEG$(Q$,I,1))<45
 )+(ASC(SEG$(Q$,I,1))>57)+(SE
 G$(Q$,I,1)="/")THEN 140 ELSE
  160
>140 PRINT "X-"
>150 GOTO 100
>160 NEXT I
>170 NBR=VAL(Q$)
>180 PRINT "OK"
>190 GOTO 100
```

If it's not a "number", depending on how thorough you want to be, you might add an additional print line saying, "NOT A NUMBER", in line 150 before sending it back to the input line. After going through this section you can confidently convert this to a numeric value without fear of the computer giving an error message. Now we can go on to test and "structure" the number to our specifications.

Assuming NBR is our number, the following will convert any number containing decimals to a straight integer value (everything to the left of a decimal).

```
>100 NBR=INT(NBR)
```

Using this value and adding some additional lines, you can round to the nearest whole number, tenth, hundreth, etc., using the following statements:

```
>100 INPUT "NBR ":NBR
>110 NBR1=INT(NBR+.5)
>120 NBR2=INT((NBR*10)+.5)/10
>130 NBR3=INT((NBR*100)+.5)/1
 00
>140 CALL CLEAR
>150 PRINT NBR;NBR1;NBR2;NBR3
>160 GOTO 100
```

Assuming that you have verified that you have a number or that you have used the "numeric" input, the following subroutine combines several of the above and will round to the nearest hundredth, add the necessary zeros and decimals, and right justify all numbers in a field 10 characters wide. This is the most common arrangement for dollars and cents data.

```
>100 REM GET NUMBER AS NR
>110 INPUT NR
>120 GOSUB 150
>130 PRINT NR$;"  ";LEN(NR$)
>140 GOTO 110
>150 REM SUBROUTINE
>160 SP=0
>170 SP$="          "
>180 NR$=STR$(INT((NR*100)+.5
 )/100)
>190 FOR I=1 TO LEN(NR$)
>200 IF ASC(SEG$(NR$,I,1))<>4
 6 THEN 220
>210 SP=I
>220 NEXT I
>230 IF SP=0 THEN 290
>240 IF SP=LEN(NR$)-1 THEN 27
 0
>250 NR$=(SEG$(SP$,1,10-LEN(N
 R$)))&NR$
>260 GOTO 300
>270 NR$=(SEG$(SP$,1,9-LEN(NR
 $)))&NR$&"0"
```

```
>280 GOTO 300
>290 NR$=(SEG$(SP$,1,7-LEN(NR
 $)))&NR$&".00"
>300 RETURN
```

It would seem after all this testing that all possible problems would have been eliminated. In fact, you could still error out the above routine by entering numbers with commas or a string with two periods in it. The double periods could be locked out with additional statements if you think it might occur. The computer will have to check for the commas. Even a string won't prevent this problem. In spite of all the above, even if you have technically correct data that can be structured and manipulated, the data still might be wrong or illogical.

Logic - Testing Range. After each input, you should consider whether the response can be tested for logic. If you wrote a program that created a data file on all bills due, and the date each one was due, the date would be a critical part of the program. You could give an example, showing that the input should look like "012383"; however, a person might invert a couple of numbers and enter "013283". If you're relying on that date, you may be in trouble. You might want to consider putting in a series of IF statements to check the values for month, day and year.

```
100 INPUT "DATE ":DT$
110 DT1=VAL(SEG$(DT$,1,2))
120 DT2=VAL(SEG$(DT$,3,2))
130 DT3=VAL(SEG$(DT$,5,2))
140 IF (DT1<1)+(DT1>12) THEN 100
150 IF (DT2<1)+(DT2>31) THEN 100
160 IF DT3<>83 THEN 100
170 PRINT "OK-";DT$
180 GOTO 100
```

The above simply checks for most legal dates within the current year. In some programs, you may start out by entering the current date. Then you could check to see if an invoice date was later than the current date. It might not be impossible, but it would be unlikely. Often these types of error messages don't reject information, but may ask you to verify it again to make sure it's correct. On a personal checkbook program you might put in a logic test to see if any one check exceeded $1000.00, or some other figure that would be higher than what you might normally write. It helps eliminate the possibility of an extra zero.

Logic - Check Digits. If you get involved in writing any program using account numbers, inventory numbers, part numbers, etc., you may begin to experience problems with transposing numbers. Unless you compare your inputted account number against your entire list of possible account numbers, how would you know if you transposed a number? Do you want to sit at the keyboard after each input while it compares the figure to 100 or 200 possible accounts? Using something called a "check digit" when building these programs can solve this problem.

A check digit is a single digit number which is added as the last number of an account number, or other number that you want to validate, to insure that you haven't transposed numbers during input. Following is a test program and a discussion of when to use it and how it works:

```
>100 CALL CLEAR
>110 INPUT "RAW NR: ":NR
>120 A=NR
>130 GOSUB 210
```

```
>140 NR=A
>150 PRINT "NEW NR: ";NR
>160 PRINT ::
>170 INPUT "ANY NR: ":ANY
>180 GOSUB 310
>190 GOTO 170
>200 REM CALC CHECK DIGIT
>210 T1=0
>220 N=LEN(STR$(A))
>230 FOR I=1 TO N
>240 N1$=SEG$((STR$(VAL(STR$(
 2^(N+I))))),1,1)
>250 T1=T1+(((VAL(STR$(VAL(N1
 $)))))*(VAL(SEG$(STR$(A),I,1
 ))))
>260 NEXT I
>270 CK=T1-(INT(T1*.1)*10)
>280 A=(A*10)+CK
>290 RETURN
>300 REM  COMPARE CHECK DIGIT
>310 A=VAL(SEG$((STR$(ANY)),1
 ,LEN(STR$(ANY))-1))
>320 CKC=VAL(SEG$((STR$(ANY))
 ,LEN(STR$(ANY)),1))
>330 GOSUB 210
>340 CKD=VAL(SEG$((STR$(A)),L
 EN(STR$(A)),1))
>350 IF CKC=CKD THEN 380
>360 PRINT "NOT VALID"
>370 GOTO 390
>380 PRINT "VALID"
>390 RETURN
>RUN
```

As an example, let's say you're setting up a data file which is going to contain 5 digit account numbers. These could be customers, prospects, people in your neighborhood, church members, etc. As you are building this file, or each time you add a name after it is built, someone will have to decide what account number to assign. Your program may simply add them sequentially or you may individually assign them. One way or the other, you'll have a five digit number to start with. This would equate to the RAW NR variable shown in line 110 above. It's much easier to explain this if you run the program as we go, so if you haven't entered it yet, we suggest you do so.

If we enter the number 72591, the program prints back a NEW NR of 725912. This six digit number is the number you should use on all of your printouts, reports, lists, and in the data file itself. Now when you go back into your data file to make changes in addresses, dates, amounts, or whatever, one of the things the computer will ask for is the account number. On our sample program this is the equivalent of the question ANY NR which is now appearing. Answer this question by entering the proper number of 725912. The computer agrees that it's a VALID number. Now transpose the 2 and the 5 (2nd & 3rd) numbers and enter 752912. Even without checking against a list of existing numbers, the computer knows that this number is NOT VALID. Experiment yourself with the numbers and you'll see how this eliminates transpositions. By the time you find another one that comes up valid, the numbers will have to be so mixed up that it couldn't possibly be a typing error. Here's how it works.

To create the check digit, which will eventually be the last digit of our account number, the computer performs a calculation involving every number of your RAW number. It starts in line 220 by setting a value for N based on the length of the RAW number (in this case 5). It then goes through a FOR NEXT loop and evaluates each number. For the first number, it derives a figure of 2 ^ 6 (two to the 6th power) or 64; and it converts the first digit of that number (6) to N1$. This all takes place in line 240. In line 250, this value of 6 is multiplied times

the first digit in the account number (which is 7); and the product (42) is added to the previous value of T1. This is repeated for each of the five numbers: increasing the exponential figure in line 240 each time; taking the first digit of that figure times the 2nd, 3rd, 4th, and 5th numbers; and adding each of their products to T1. After the loop is finished, in lines 270 and 280, the computer selects the last digit of T1 and adds it to the original five digit number as a "check digit". To see these calculations, add the following to lines to the example program:

```
>245 PRINT SEG$(STR$(A),I,1);
 2^(N+I);TAB(10);N1$;
>255 PRINT ((VAL(STR$(VAL(N1$
 )))))*(VAL(SEG$(STR$(A),I,1)
 )));TAB(20);T1
```

The section that checks the input starts in line 300 and just reverses the process. When you input a six digit number, it extracts the first five digits (line 310) and gets the check digit (line 320). It then performs the normal check digit calculation on the first five digits by going back through the first subroutine. When it arrives at a figure, it compares that with the check digit entered. If they're not equal, it's not a valid number. This is an extremely useful tool and well worth building into a program during the creation of your original data file.

Checking Computer Data. We've devoted a lot of space to keyboard input because, between the computer and a human, the human is far less predictable; however, we need to do a little checking on computer generated data as well. There are certain statements which are used, and actions that take place within a program, that almost always require validation either just preceding or just after them.

Anytime you have a statement which creates a row and/or column number that will subsequently be used in a CALL HCHAR or CALL VCHAR statement, you must be sure that the number is within the acceptable range. Many programs requiring movement have a current location stored as something like ROW and COLUMN. At some point the programs will add to or subtract from this figure to get a new row and column. The same thing holds true for all of your CALL commands such as CALL SOUND, CALL GCHAR, CALL CHAR, CALL COLOR. These all require some variable within the parenthesis following the command. Unless that variable is controlled, meaning that it was assigned by a DATA and READ statement, or through a RND (Random) command with appropriate limits, the variables should be tested prior to the command. Two other critical points are just prior to the ON ___ GOTO command and ON ___ GOSUB command. If you have four line numbers following the command, the variable must be between 1 and 4. Last, consider it when a variable is created that will be used as a subscript for an array, such as the N in FMY$(N).

While it's impossible to give you a definite rule that will work for all programs, we can give you a series of questions, as we did above, that you can ask yourself anytime a variable is created or modified.

1. What do I do if it's equal to Y (another variable or a fixed limitation like 32 for maximum column)?
2. What if it's less than Y?

3. What if it's greater than Y?
4. Are there both upper & lower limits.?
5. If it's within legal limits, what are the possibilities?

Again, the answer you give yourself may be that, "It doesn't apply in this case". The important thing is that you consider all possible results of a calculation on a variable and that you have a solution for every possible result. The formats for testing for the first three possibilities are straight forward. They are:

1. IF X=Y THEN ____
2. IF X<Y THEN ____
3. IF X>Y THEN ____

Of course, if the response is the same, many of these can be combined to form statements like IF X<=Y. In case you haven't already picked it up from some of the programs, testing for limits is usually done using the logic capability of the IF statement. This is a powerful tool and may require a little explanation.

In the above statements, the computer has evaluated the expression following the word IF and prior to the word THEN. If this evaluation is not true (false) then it sets a little marker in its memory to "zero". If it's true it sets the marker to "1". When it sees the THEN statement it checks the value of its marker. If the value is "1" it will go to the line specified after the THEN statement, if it's "Ø", it'll look to see if there is an ELSE statement. If there is an ELSE statement it will go to that line; otherwise, it will go on to the statement following the IF-THEN statement. The important thing to remember is that it will evaluate not just one, but all of the expressions between the IF and THEN to set these markers. Each of these expressions can be added or multiplied, and the combination of 1's and Ø's will be evaluated. A zero is treated as false and anything greater than zero is true. To explain this relationship, let's set up an example using R for Row and C for Column on a screen display.

```
>100 INPUT "ROW,COL? ":R,C
>110 IF (R>0)*(R<25)*(C>0)*(C
 <33) THEN 140
>120 PRINT "NOT GOOD"
>130 GOTO 150
>140 PRINT "GOOD"
>150 PRINT
>160 GOTO 100
```

The above statement will effectively eliminate any out of bounds value for row or column. Use the following inputs (or develop your own) as a test.

VALID -	2,12	4,16	7,32
NOT VALID -	2,Ø	5,33	Ø,32

In the above, if all conditions are right, each of the four relationships have a value of "1". The total value of the relationship is thus: 1*1*1*1=1 and the statement is "GOOD". If any one of the four statements is not true, a "Ø" will become part of the expression so that: 1*1*Ø*1=Ø. All that's required is one incorrect answer to make it false. You must be careful with your operators. If the above used a (+) sign in between, it would not properly operate. In that case, if one relationship was wrong, the result would be: 1+1+Ø+1=3. This would still result in a positive reaction since the total is greater than 1. Using the multiplication sign means that all conditions must be met for it to be true. A plus sign means

that any one of them is enough to make it true. Depending on how you structure the individual expressions, the choice between multiplication and addition is yours. In the first example we checked for all positive relationships and in the second we looked for any negative condition. Change line 11Ø as follow to see the difference:

```
>11Ø IF (R<1)+(R>24)+(C<1)+(C
 >32) THEN 12Ø ELSE 14Ø
```

Evaluating Options. Computers really are ignorant. The IF statements are the decision makers of a program and those statements are a product of the programmer. In the real world when we have a decision to make, we normally gather all of the facts and evaluate them before making our choice. Some possibilities are eliminated very quickly as being definitely wrong while others are definitely right; however, there are usually a lot of other choices which fall into the "gray" area. Assuming that our program has effectively selected some possible answers, "who's going to make the final decision as to which one to use?" Don't assume that a program will do anything! You have to think through each portion and tell the computer how to react. The following little program clears the screen and then starts randomly filling up an 8 X 8 area in the middle of the screen with (*).

```
>1ØØ RANDOMIZE
>11Ø CALL CLEAR
>12Ø ROW=12
>13Ø COL=16
>14Ø A=-3
>15Ø B=+3
>16Ø CALL HCHAR(ROW,COL,42)
>17Ø RR=INT((B-A+1)*RND)+A
>18Ø CC=INT((B-A+1)*RND)+A
>19Ø ROW=ROW+RR
>2ØØ COL=COL+CC
>21Ø IF (ROW<8)+(ROW>15)THEN
 24Ø
>22Ø IF (COL<12)+(COL>19)THEN
 24Ø
>23Ø GOTO 16Ø
>24Ø GOTO 17Ø
>RUN
```

At first glance this arrangement appears quite logical. It sets a starting row and col; gets a random figure between +3 and -3 and adds it to the current figure; if it's outside of the 8 X 8 square it goes back to the random statement to try again. However, when you run the program, it very quickly ceases to fill the block. What's lacking? Add the following lines to see the problem.

```
>2Ø5 PRINT "  ";ROW,COL;
>23Ø PRINT
>235 GOTO 16Ø
>24Ø PRINT "**"
>245 GOTO 17Ø
```

At first we get acceptable numbers, but very rapidly it starts getting out of "range". Although the computer doesn't "error out", the numbers it's generating are getting farther and farther away from the center of the screen. If you run it long enough it may also work its way back in, but there's no way of knowing that for sure. Now make the following adjustments:

```
>2Ø5
>23Ø GOTO 16Ø
>235
>24Ø ROW=12
>245 COL=16
>25Ø GOTO 17Ø
```

Run this program several times. This will eventually fill it up; however, as it approaches full, it seems to be taking quite a bit of time before it hits another blank spot. See if you can work on it and find still a better way. This is a typical example of the kind of problem that so often creeps into a program. If a program keeps "bombing", and you keep adding more validation to prevent it, by trial and error you'll eventually get enough IF statements in it so that the computer will always make a "valid" choice. In order to guide the computer to the "best" choice, you're going to have to THINK beyond the obvious. As we said before, "the computer is ignorant". Given five or ten "legal" choices, the computer cannot logically figure out which is "best". Only you can THINK!

```
**********************
*    TABLE OF 12'S   *
*      V-PN731KB     *
*    BY T CASTLE     *
**********************
```

DESCRIPTION. Remember the flash cards. Here's a program that the youngsters can while away the hours with working on the multiplication and division tables from one to twelve.

The program opens with a "Menu" to set the parameters of the program. You select: random or sequential order; multiplication or division; the maximum number to appear in the question and; the minimum number to appear in the question. For multiplication only, you can select a single number that you want to test. After selection the computer will build the necessary array of all possible combinations within the range specified.

Screen display is gray, with all numbers shown in black on yellow strips. Each yellow strip is bordered with red. The equation, minus the answer, appears across the bottom of the screen. Above it, there is a "diamond" consisting of 8 numbers, only one of which is the correct answer. A red block moves continuously around the diamond. Hitting any key when the red block is by the correct answer scores points. In the upper left portion of the screen your score is shown and, below that, the maximum points you could obtain. In the upper right the number of wrong answers is shown and, below that, the number of questions remaining on each cycle. The red block will go around 5 times (passing the correct answer) before the equation changes, unless the student selects the right answer, in which case it changes immediately.

NOTES. The layout of the program is our traditional subroutine method, with the entire sequence shown in lines 17Ø through 29Ø. Because we have several FOR-NEXT loops that go around the diamond we have set up a special data statement in line 56Ø which designates the row and column numbers for the left side of each of the 8 answer blocks. The nested FOR-NEXT loops (1Ø7Ø-115Ø) build the initial array of all combinations and, if random is selected, the array is simply shuffled (like a deck of cards) in lines 116Ø-125Ø.

The main control loop of the program begins in line 128Ø and runs through 2Ø8Ø. This consists of four (4) nested FOR-NEXT LOOPS: J=1 TO REPS (128Ø) controls number of questions asked; CYC=1 TO 5 (175Ø) controls the five passes around the diamond and RESTORES data statement; CIR=1 to 8 controls movement of the red block around diamond; and ASK=1 TO 4 (181Ø) permits 4 call keys at each location to see if student has hit a key.

```
100 REM  *****************
110 REM  * TABLE OF 12'S *
120 REM  *****************
130 REM
140 REM  BY T CASTLE
150 REM  AMLIST V-PN731KB
160 REM
170 REM  INITIAL VARIABLES
180 GOSUB 310
190 REM  MENU
200 GOSUB 770
210 REM  BUILD ARRAY
220 GOSUB 1080
230 REM  BUILD DISPLAY
240 GOSUB 510
250 REM  PRINTS QUES & ANS
260 GOSUB 1280
270 REM  CLOSE OUT
280 GOSUB 2330
290 GOTO 180
300 REM  SETS VARIABLES
310 CALL CLEAR
320 RESTORE 350
330 DIM QS$(144)
340 RANDOMIZE
350 DATA 128,000000000000FFF
F,130,0101010101010101,131,8
080808080808080
360 DATA 136,0,62,000010007C
001,129,FFFF,135,FFFFFFFFFFF
FFFFF
370 FOR I=1 TO 7
380 READ A,B$
390 CALL CHAR(A,B$)
400 NEXT I
410 CALL SCREEN(4)
420 CALL COLOR(13,7,15)
430 CALL COLOR(14,2,12)
440 CALL COLOR(3,2,1)
450 CALL COLOR(4,2,1)
460 CALL COLOR(8,2,1)
470 DATA 0,0,0,0,0
480 READ SCORE,WRONG,INC,MXA
,RMA
490 RETURN
500 REM  BUILD DISPLAY
510 CALL CLEAR
520 CALL SCREEN(15)
530 CALL COLOR(3,2,12)
540 CALL COLOR(4,2,12)
550 CALL COLOR(8,2,12)
560 DATA 7,19,11,22,15,19,19
,14,15,9,11,6,7,9,3,14
570 FOR I=1 TO 8
580 READ A,B
590 CALL HCHAR(A,B,130)
600 CALL HCHAR(A-1,B+1,128,3
)
610 CALL HCHAR(A+1,B+1,129,3
)
620 CALL HCHAR(A,B+4,131)
630 CALL HCHAR(A,B+1,136,3)
640 NEXT I
650 DATA 1,3,128,5,1,25,128,
5,2,2,130,1,2,8,131,1,2,24,1
30,1,2,30,131,1
660 DATA 3,3,129,5,3,25,129,
5,2,3,136,5,2,25,136,5,22,9,
128,15
670 DATA 23,8,130,1,23,24,13
1,1,24,9,129,15,23,9,136,15
680 DATA 4,3,128,5,4,25,128,
5,5,2,130,1,5,8,131,1,5,24,1
30,1,5,30,131,1
690 DATA 6,3,129,5,6,25,129,
5,5,3,136,5,5,25,136,5
700 FOR I=1 TO 25
710 READ A,B,C,D
720 CALL HCHAR(A,B,C,D)
730 NEXT I
740 CALL HCHAR(2,25,88)
750 RETURN
760 REM  MENU-INSTRUCTIONS
770 CALL CLEAR
780 PRINT ::::"     INSTRUCTO
R OPTIONS":::
790 INPUT "RANDOM/SEQUENTIAL
 (R/S)? ":Q$
800 IF (Q$="R")+(Q$="S")THEN
 810 ELSE 790
810 ORD$=Q$
820 PRINT ::
830 INPUT "MULTIPLY/DIVIDE
 (M/D)? ":Q$
840 IF (Q$="M")+(Q$="D")THEN
 850 ELSE 830
850 TYP$=Q$
860 PRINT ::
870 INPUT "MAX NUMBER IN QUE
STION?  ":Q
880 IF (Q<1)+(Q>12)THEN 870
890 MXAN=Q
```

```
900 PRINT ::
910 INPUT "MIN NUMBER IN QUE
STION?  ":Q
920 IF (Q<1)+(Q>12)THEN 910
930 MNAN=Q
940 IF TYP$="D" THEN 1030
950 PRINT ::
960 INPUT "SPECIFY NUMBER TE
ST(OR O)":Q
970 IF (Q<0)+(Q>12)THEN 960
980 IF Q=0 THEN 1030
990 SPL=Q
1000 SPH=Q
1010 REPS=MXAN-MNAN+1
1020 GOTO 1060
1030 REPS=((MXAN-MNAN)+1)^2
1040 SPL=MNAN
1050 SPH=MXAN
1060 RETURN
1070 REM  SELECTS NUMBERS
1080 FOR I=SPL TO SPH
1090 FOR J=MNAN TO MXAN
1100 K=K+1
1110 QS1$=SEG$(("   "&STR$(I
)),(3+LEN(STR$(I)))-2,3)
1120 QS2$=SEG$(("   "&STR$(J
)),(3+LEN(STR$(J)))-2,3)
1130 QS$(K)=QS1$&QS2$
1140 NEXT J
1150 NEXT I
1160 IF ORD$="S" THEN 1260
1170 FOR I=1 TO INT(REPS*1.5
)
1180 J=INT(REPS*RND)+1
1190 HOLD1$=QS$(J)
1200 J1=INT(REPS*RND)+1
1210 IF J1=J THEN 1200
1220 HOLD2$=QS$(J1)
1230 QS$(J)=HOLD2$
1240 QS$(J1)=HOLD1$
1250 NEXT I
1260 RETURN
1270 REM  PRINTS QUES&ANS
1280 FOR J=1 TO REPS
1290 Q1$=SEG$(QS$(J),1,3)
1300 Q2$=SEG$(QS$(J),4,3)
1310 ANS=VAL(Q1$)*VAL(Q2$)
1320 TCD=88
1330 IF TYP$="M" THEN 1380
1340 HLD$=Q1$
1350 Q1$=SEG$(("   "&STR$(AN
S)),(3+LEN(STR$(ANS)))-2,3)
1360 ANS=VAL(HLD$)
1370 TCD=62
1380 MSG$="2308"&Q1$
1390 GOSUB 2100
1400 CALL HCHAR(23,13,TCD)
1410 MSG$="2314"&Q2$
1420 GOSUB 2100
1430 CALL HCHAR(23,19,61)
1440 CALL HCHAR(23,21,63)
1450 IF ANS>9 THEN 1490
1460 LOW=1
1470 HIG=18
1480 GOTO 1510
1490 LOW=ANS-8
1500 HIG=ANS+9
1510 RESTORE 560
1520 CK=INT(8*RND)+1
1530 FOR I=1 TO 8
1540 IF I<>CK THEN 1570
1550 L=ANS
1560 GOTO 1590
1570 L=INT((HIG-LOW+1)*RND)+
LOW
1580 IF L=ANS THEN 1570
1590 J$=SEG$(("   "&STR$(L))
,(3+LEN(STR$(L)))-2,3)
1600 READ A,B
1610 IF CK<>I THEN 1640
1620 SVA=A
1630 SVB=B
1640 MSG$="9900"&J$
1650 GOSUB 2100
1660 NEXT I
1670 RMA=(REPS)-J
1680 RMA$=SEG$(("     "&STR$
(RMA)),(5+LEN(STR$(RMA)))-4,
5)
1690 MSG$="0524"&RMA$
1700 GOSUB 2100
1710 MXA=MXA+INT((1.10*ANS)+
.5)
1720 MXA$=SEG$(("     "&STR$
(MXA)),(5+LEN(STR$(MXA)))-4,
5)
1730 MSG$="0502"&MXA$
1740 GOSUB 2100
1750 FOR CYC=1 TO 5
1760 RESTORE 560
1770 FOR CIR=1 TO 8
```

```
1780 READ A,B
1790 CALL HCHAR(A+1,B+2,135)
1800 CALL SOUND(-100,1500,10
)
1810 FOR ASK=1 TO 4
1820 CALL KEY(3,KY,ST)
1830 IF ST=0 THEN 2020
1840 IF (A<>SVA)+(B<>SVB)THE
N 1980
1850 FOR SND=1 TO 5
1860 CALL SOUND(50,1300,1)
1870 CALL SOUND(50,1100,1)
1880 NEXT SND
1890 INC=(1.10*ANS)-((CYC-1)
*(.10*ANS))
1900 SCORE=INT(SCORE+INC+.5)
1910 SCORE$=SEG$(("     "&ST
R$(SCORE)),(5+LEN(STR$(SCORE
)))-4,5)
1920 MSG$="0202"&SCORE$
1930 GOSUB 2100
1940 DATA 4,8,5
1950 RESTORE 1940
1960 READ ASK,CIR,CYC
1970 GOTO 2020
1980 WRONG=WRONG+1
1990 MSG$="0226"&SEG$(("    "
&STR$(WRONG)),(4+LEN(STR$(WR
ONG)))-3,5)
2000 GOSUB 2100
2010 CALL SOUND(200,110,10)
2020 CALL HCHAR(A+1,B+2,129)
2030 NEXT ASK
2040 NEXT CIR
2050 NEXT CYC
2060 GOSUB 2250
2070 NEXT J
2080 RETURN
2090 REM  PRINT ROUTINE
2100 LMSG=LEN(MSG$)-4
2110 ROW=VAL(SEG$(MSG$,1,2))
2120 COL=VAL(SEG$(MSG$,3,2))
2130 FOR PM=1 TO LMSG
2140 MSG=ASC(SEG$(MSG$,4+PM,
1))
2150 IF MSG<>32 THEN 2170
2160 MSG=136
2170 IF ROW=99 THEN 2200
2180 CALL HCHAR(ROW,COL+PM,M
SG)
2190 GOTO 2210
2200 CALL HCHAR(A,PM+B,MSG)
2210 NEXT PM
2220 CALL SOUND(-50,1000,15)
2230 RETURN
2240 REM  ERASE ROUTINE
2250 RESTORE 560
2260 FOR ER=1 TO 8
2270 READ EA,EB
2280 CALL HCHAR(EA,EB+1,136,
3)
2290 NEXT ER
2300 CALL HCHAR(23,9,136,15)
2310 RETURN
2320 REM  EVALUATES
2330 M$(5)=" MATH GENIUS! "
2340 M$(4)=" OUTSTANDING! "
2350 M$(3)="  GOOD JOB!   "
2360 M$(2)=" KEEP TRYING! "
2370 M$(1)=" ASK FOR HELP "
2380 A=INT((10*(SCORE/MXA))-
5)
2390 IF A<1 THEN 2430
2400 IF (SCORE/MXA)>.95 THEN
 2410 ELSE 2440
2410 A=5
2420 GOTO 2440
2430 A=1
2440 MSG$="2308 "&M$(A)
2450 CALL COLOR(8,2,1)
2460 CALL HCHAR(2,25,136)
2470 FR=110
2480 FOR SND=30 TO 1 STEP -1
2490 FR=FR+50
2500 CALL SOUND(-50,FR,SND)
2510 NEXT SND
2520 GOSUB 2590
2530 MSG$="1110"&"HIT ANY KE
Y"
2540 GOSUB 2590
2550 CALL KEY(3,KY,ST)
2560 IF ST=0 THEN 2550
2570 RETURN
2580 REM  STRAIGHT MESSAGE
2590 ROW=VAL(SEG$(MSG$,1,2))
2600 COL=VAL(SEG$(MSG$,3,2))
2610 FOR I=1 TO LEN(MSG$)-4
2620 CALL HCHAR(ROW,COL+I,AS
C(SEG$(MSG$,I+4,1)))
2630 NEXT I
2640 RETURN
```

HAPPY COMPUTING!

CHAPTER TEN

Condensing & Refining

GENERAL. The first thing that most people think of when we mention condensing a program is cutting down the number of lines that they need to type in. While this is certainly one form of condensing, it's by no means the sole topic of this discussion. The idea is to get the absolute maximum out of your program in terms of what it will do, how fast it will do it, and how many lines of code it will take to get the job done. Actually the number of lines it takes should be a secondary consideration, provided that it operates at peak efficiency as far as the user is concerned.

Like debugging, this is not a single step in the development of a program that occurs only at the end. Rather, it's a continuous process which should begin even before you begin to code in a program; it's continued throughout the coding stage; and may or may not be pursued further after the basic program is operating. As with many other aspects of programming, condensing and refining often means compromises. Removing "range" testing, check digits, and other forms of validity testing will definitely reduce the number of lines in a program; however, it simply shifts the burden of accuracy back to the user and reduces the power of the computer to make decisions. A decision to use string variables to represent numbers will often result in a definite savings in memory; however, the time required for processing these "numbers" increases since the computer must now convert them back to numeric variables. What the programmer hopes to achieve is the best possible balance between all of these factors. This optimum arrangement isn't always obvious during program development.

At the end of this chapter there is a condensed and refined version of one of the early programs in this manual, the Building Blocks program. This program, as originally written, complete with documentation, consisted of some 448 lines of code. The attached version does exactly the same thing, in some cases even better (key response is improved) with only 129 lines of code. Looking at just this factor, that's a reduction of 71.2%. Since it's similar to what you're going to encounter in many of your own programs, we're going to show you, step-by-step, how this was done. Regardless of whether you're interested in children's programs or not, read the description accompanying the Building Blocks program so that you have a general understanding of what it does so that you can follow the discussion about to take place. If you want to see it in operation and you haven't already entered it, we suggest you enter the short version. Earlier in our discussions we talked about the use of subroutines to lay out our general program and how each of these would then be written almost as small separate programs. Let's go back now and look at this process again, as it was used in the Building Blocks program.

Subroutines. This program was completely developed using the subroutine theory. We took each thing that we knew had to be done somewhere and simply set it up as a single subroutine (GOSUB). It followed a logical pattern of necessity, though not necessarily of sequence. The logic was as follows:

1. Before we could do anything we obviously needed colors and character definitions so these were set and defined first (310-590).
2. Next we had to combine the characters to create the nine shapes we wanted to print (600-1220).
3. We then needed a stationary, blank display to work from (1230-2130).
4. Next we asked for user input (2140-3700).
5. We evaluated and printed the shape based on user input (3710-4570).
6. Finally, we developed a series of subroutines to route the program through the various routines (160-300).

This step-by-step process made program development rather simple; however, is it the best possible arrangement? Have you ever worked one of those "brain teasers" involving pegs, toothpicks, or rings. Isn't it amazing how simple things are once someone shows you the answer. Programming is much the same and, as we all know, hindsight is a great thing. Having once written the program it was obvious that we could have eliminated a lot of lines by simply rearranging the sequence. We could drop all of the lines from 160 to 300; move the subroutines that print the shapes (600-1220) to the end of the listing; move the subroutine that prints the question (2140-2530) to the end; and finally, remove all the RETURN statements from the remaining portions. With very few other modifications, the program could define variables, build a display, ask all questions, print the shape, and GOTO (not GOSUB) directly back to the questions. This was the first step taken on the Building Blocks program and we were able to eliminate about 25 lines.

If brevity in line numbering and less coding is your goal, review every program after it's written and mark down how many times your program GOSUB's to a specific routine. If it's only once, you can remove it and place it directly where it belongs in the program. On the reverse side, scan your program for sections which are similar and <u>not</u> in subroutines. If you find several sections which consist of essentially the same commands, consider putting them into a subroutine at the end of the program. In the Memory Jogger program, notice the similarity between the statements in line 520-550 and 1620-1650. Both are running through FOR - NEXT loops, except one used an "I" variable and the other a "J" variable. With a little effort these could have been combined into one subroutine.

Before we leave subroutines let's also discuss how fast the subroutine operates. It's a popular theory that using these will slow down the particular operation because of the branching required. Please enter the following test:

```
>100 CALL CLEAR
>110 CALL SOUND(50,260,0)
>120 FOR I=1 TO 500
>130 GOSUB 170
```

```
>14Ø NEXT I
>15Ø CALL SOUND(5Ø,26Ø,Ø)
>16Ø STOP
>17Ø PRINT "TEST ";I
>18Ø RETURN
```

For a completely accurate test you would have to run this ten or twenty times and perhaps increase the repetitions. For our purposes, once should be sufficient. The above program provides a "beep" at the beginning and end of the FOR - NEXT statement so you can time the movement. Using a GOSUB in line 13Ø we timed it at 1 minute 52 seconds. If you replace the GOSUB statement in 13Ø with the statement in 17Ø, you'll see the same results on the screen without use of a GOSUB. We timed this at 1 minute 5Ø seconds. We could be slightly off, but it would appear that the time lost is less than 2%. In 5ØØ passes it resulted in only 2 seconds difference and for a single or occasional reference in a working program, it would seem insignificant.

DATA Lines. In console basic we are limited to one command, such as PRINT, DATA, INPUT, etc, per numbered line. Still, these three offer some potential for consolidation of individual statements. To look at DATA statements, let's pick on the Building Blocks program again. In lines 41Ø-48Ø, we've listed only 2 character definitions per data line. This was for the convenience of the user who is copying it in. It makes it easier to see whether you've forgotten a character since the lengths are the same. As our next step in condensing the BB program, we combined all of the information in these 8 lines on three long lines. Likewise, all of the data from 183Ø-193Ø (11 lines) was combined on 2 continuous long data statements. The first move wound up in our finished version; however, as you will later see, the second group was no longer needed.

We've timed READ statements from both short and long data lines and the difference between the two appears to be negligible. In general, you can use four complete screen lines. At the end of the 4th line you'll get a buzzer and the cursor will move no further. We say "in general" because, even though you can type in four lines, some data statements that fill four lines will come back with the error message "LINE TOO LONG". With single digit data elements, you can go only to the first entry on the 4th line. Attempting to enter more will cause the error message and you'll have to reenter all of the line again. With a number of short data elements, stick to a three line limit. The computer will accept the following line; however, add one more item and it will error out.

```
>3Ø2 DATA 1,1,1,1,1,1,1,1,1,1
 ,1,1,1,1,1,1,1,1,1,1,1,1,1,1
 ,1,1,1,1,1,1,1,1,1,1,1,1,1,1
 ,1
```

There are also some instances where you can actually enter more than four lines of data. Since it's a bit complicated, it's covered as a separate part of this chapter.

PRINT & INPUT Statements. Since this program involved graphics, we weren't scrolling information onto the screen and we therefore had no PRINT or INPUT statements; however, in other programs these will provide you with a place for "cutting" lines, so we'll digress for a moment and discuss these.

For PRINT statements, the colon (:) is a great aid. Printing a phrase and scrolling it to the middle of the screen could be accomplished as follows:

```
>100 CALL CLEAR
>110 PRINT "HAPPY BIRTHDAY"
>120 FOR I=1 TO 10
>130 PRINT
>140 NEXT I
>150 GOTO 150
```

But why not:

```
>100 CALL CLEAR
>110 PRINT "HAPPY BIRTHDAY"::
 ::::::::
>120 GOTO 120
```

Or better yet:

```
>100 PRINT ::::::::::::"HAPP
 Y BIRTHDAY":::::::::
>120 GOTO 120
```

This reduces 6 statements in the first example to one in the last. To clear the screen, all you need is a combination of colons and actual lines printed that totals 24. Scrolling information off the screen is not as fast as a CALL CLEAR, but it does accomplish the goal. Here's another example of how to line up your characters for multiple print lines.

```
>100 PRINT "THIS IS ONE WAY"
>110 PRINT "TO PRINT THREE LI
 NES OF IN-"
>120 PRINT "FORMATION TO THE
 SCREEN"
>RUN
```

```
>100 PRINT "THIS IS ONE WAY
            TO PRINT THREE LI
 NES OF IN- FORMATION TO THE
 SCREEN"
>RUN
```

Notice how the "T" in "THIS", "T" in "TO", and "F" in "FORMATION" are directly under each other. Lining up the first letters of each line (fill in the difference with spaces) can give you up to four print lines in one PRINT statement. This will hold true whether you're line numbers are 1, 2, or 3 numbers long, like 110, 1100, 20, etc. The second example above also consumes 8 bytes less of memory. Of course, the colon separator would also work and permit more than four lines if they are shorter. The program following actually prints on five separate lines with one statement.

```
>100 PRINT "THIS IS ONE WAY":
 "TO PRINT THREE LINES OF IN-
 ":"FORMATION TO THE SCREEN":
 "AND":"SO IS THIS"
```

The same type of technique can also be used following the INPUT statement. This could give you a couple lines of input instructions and the input statement itself on one line. For instance:

```
>100 INPUT "HI
            MY NAME IS JOHN
            WHAT'S YOURS?":Q$
```

Validation Statements. Let's go back now to the Building Blocks program. Having rearranged the subroutines and having eliminated some, we next looked at the validation statements to find still more areas which could use some refinement. Look specifically at the section from 3710 to 4570. In a program like this we had a choice of five colors and nine shapes (sizes). By the time it got to this point in the program the value of A1 was the ASC code for the letters A through E (four colors & clear), and A2 represented the ASC code for the shapes and sizes. In order to test

our theory of printing it to the screen, we just allowed one option at first, i.e. A1=65 and A2=65. We wrote the necessary IF statement and tested it. When we were sure it worked, we simply added more IF statements for each of the possibilities. It worked fine, it was neat, and it was easy to read, so we went no further. Since this exercise is concerned with condensing, we simply replaced these statements with ON __ GOTO __ statements. Looking at the numbers following the IF statements, you can see they are all sequential, so converting them to integers from 1-4 and 1-9 was quite simple. Line 3780 was changed to:

```
>378Ø ON A1-64 GOTO 383Ø,385Ø
 ,387Ø,389Ø,391Ø
```

Line 4Ø8Ø was changed to:

```
>4Ø8Ø ON A2-64 GOTO 417Ø,421Ø
 ,425Ø,429Ø,433Ø,437Ø,441Ø,44
 5Ø,449Ø
```

Looking in other subroutines we found still more lines where logic statements would work as well. Lines 33ØØ-332Ø, 335Ø-337Ø, 258Ø-259Ø, and others were converted to something like:

```
>33ØØ IF (A2=65)+(A2=68)+(A2=
 71) THEN 333Ø ELSE 335Ø
```

<u>Data & Arrays.</u> After all of these consolidations, we still had about 3ØØ lines of code. It was obvious that a lot of lines were used with CALL HCHAR and still more were used setting the values for R1 (Row), C1 (Column), and CC (Color Code). Each of these was used as a starting point and sent to each individual subroutine. We considered the possibility here of converting statements to strings and sending them as one package to the subroutine that printed the shape. For instance, we could take lines 151Ø-153Ø and convert them to:

```
>151Ø DAT$="Ø321153"
```

Then, instead of using R1, C1, and CC in the subroutines from 69Ø-122Ø we could replace lines like 118Ø with:

```
>118Ø CALL HCHAR(VAL(SEG$(DT$
 ,1,2)),VAL(SEG$(DT$,3,2)),VA
 L(SEG$(DT$,5,3))+5)
```

There were about 26 places where this happened and we could have "swapped" one line for three at each point. This would reduce the program by about another 52 lines; but, we wanted major reductions. If we had stopped at this point, we would have wound up with perhaps 27Ø lines. In order to find places for further reductions we had to quit looking at four or five line segments and reconsider our whole method of handling information. Looking at the series of subroutines from 77Ø to 122Ø we noticed that all but a few of them were CALL HCHAR statements; all had the values of R1, C1, CC, with or without an adjustment (-1, 1, -5, etc); in a few there were repetitions added (line 1Ø3Ø had repetitions of three). We decided to set up one subroutine and one CALL HCHAR that could handle any situation and use DATA and READ statements to send it the adjustment. The routine looked like:

```
>5ØØØ FOR K=1 TO 8
>5Ø1Ø READ AJ1,AJ2,AJ3,AJ4
>5Ø2Ø IF AJ1=99 THEN 5Ø5Ø
>5Ø3Ø CALL HCHAR(R1+AJ1,C1+AJ
 2,CC+AJ3,AJ4)
>5Ø4Ø NEXT K
>5Ø5Ø RETURN
```

Now all we had to do was put a RESTORE and DATA statement at each point before it went to the subroutine and send them all to the same subroutine. For instance, the lines from 4210 to 4240 would look like:

```
>4210 R1=R2-1
>4220 C1=C2
>4222 RESTORE 4224
>4224 DATA Ø,Ø,1,1,1,Ø,2,1,Ø,
 1,3,1,1,1,4,1,99,Ø,Ø,Ø
>4230 GOSUB 5000
>4240 GOTO 4530
```

This changed the whole complexion of things from 4080 on. Now all of them did a GOSUB 5000 and then all of them went to 4530; thus, another opportunity presented itself. We simply changed 4080 to an ON __ GOSUB, and set up all of the data statements as subroutines.

```
>4080 ON A2-64 GOSUB 4170,421
 Ø,4250,4290,4430,4370,4410,4
 4450,4490
>4090 GOSUB 5000
>4100 A1=Ø
>4202 A2=Ø
>4204 A3=Ø
>4206 A4=Ø
>4208 RETURN
>4210 R1=R2-1
>4220 C1=C2
>4222 RESTORE 4224
>4224 DATA Ø,Ø,1,1,1,Ø,2,1,Ø,
 1,3,1,1,1,4,1,99,Ø,Ø,Ø
>4230 RETURN
(And so on thru 4520)
```

Looking at part of the start display from 1270 to 1810 we found other places that could make similar use of the same data statements. Only the values of R1,C1,CC were changed. With some slight modifications we gained a few more lines here. We were also able to eliminate all of the individual CALL HCHAR statements from 770 to 1220. A quick listing of the program showed us that although we had reduced a substantial number of lines, we still had an awful lot of RETURN and RESTORE statements. What we needed was a way to reference these deviations without having to RESTORE each time. Since we had shapes one through nine established, and the deviations for each one coded, we simply put them into an array called SET(9,30), and filled that set using four data lines (new program lines 360-390). This enabled us to remove all the GOSUB's from 4210 and to simply use a FOR-NEXT loop to build the opening display (new program lines 510-580).

Using the same philosophy as above, we took all of the other "words" such as GREEN, RED, SMALL, MEDIUM, COLOR, SHAPE, etc. and put them in arrays. Note that the conventional way to build a subscripted array is with a FOR-NEXT loop. For instance CL$ might have been built:

```
>320 DATA (words follow)
>330 FOR I=1 TO 6
>340 READ CL$(I)
>350 NEXT I
```

This would have taken 4 lines just for one array. We built all three arrays using just four lines. If you only have a few elements for each (as we have here), you can often get the job done using fewer lines by specifying the subscript in a READ statement instead of using the FOR-NEXT statement. These modifications pretty well eliminated the need for lines 2220-2480, 2650-2850, and 2960-3240. We were now down to a couple of hundred lines and we still found more places to "cut".

Look at the statements that begin in 33ØØ, 335Ø, 353Ø, 358Ø, 394Ø, 399Ø. Remember how we converted these to logic statements that looked like:

```
>33ØØ IF (A2=65)+(A2=68)+(A2=
 71) THEN 333Ø ELSE 335Ø
```

If you study what they do, you'll find that they simply determine whether the shape is small, medium, or large. In the process of rewriting the program we made this a very easy thing to determine. Since we've created an array which holds the deviations for each size and shape, we used the same array to store a number indicating its size (as the first element in the array). A zero was used to represent the smallest, a "1" for a medium, and a "2" for large shapes. After making this change, all of the CALL KEY statements remaining looked remarkably similar. About the only thing left was a small "range" test after each input and a couple of print statements. It was an easy task to just place the CALL KEY inside of a FOR-NEXT loop 69Ø-1Ø6Ø. We used an ON __ GOTO __ statement inside the loop to check range and perform any special task for each question.

Final Results. After removing some REMarks, this program was reduced from 448 lines to 129 lines. If you timed how long it took to load the program from the first sound until the computer said to STOP the recorder, load time was reduced from 1 minute 52 seconds (old way) to 1 minute even (new way). After you type RUN until you hear the first input "beep", time increased for the new program to 48 seconds. The old way took 43 seconds. The new program consumes about 7,624 bytes, while the old program used 8,5Ø4. The response to the CALL KEY was greatly improved since the first version was just slightly "sluggish". While the gains in line reduction were very substantial, all of the other gains were far less dramatic.

Adding Data Lines. We said previously that we were limited to four "roll-overs" when entering data lines and other information into long lines. This meant that you could get as many as six, sixteen digit, character codes on a line. Have you ever gotten to the point where you needed just one or two more and couldn't quite get it in four lines? There is an answer. The following program demonstrates what can be done to go to a fifth and even sixth line. Look at it first, then we'll tell you how to enter it.

```
>1ØØ CALL CLEAR
>11Ø X=1Ø
>12Ø J=X+X+X+X+X+X+X+X+X+X+X+
 X+X+X+X+X+X+X+X+X+X+X+X+X+X+
 X+X+X+X+X+X+X+X+X+X+X+X+X+X+
 X+X+X+X+X+X+X+X+X+X+X+X+X+X+
 X+X+X+X+X+X+X+X+X+X+X+X+X+X+
 X+X+X+X+X+X+X+X+X+X+X
>13Ø PRINT J
>14Ø DATA 1111111111111111,22
 22222222222222,3333333333333
 333,4444444444444444,5555555
 555555555,6666666666666666,7
 777777777777777,888888888888
 8888
>15Ø FOR I=1 TO 8
>16Ø T=T+2
>17Ø READ A$
>18Ø CALL CHAR(127+I,A$)
>19Ø CALL HCHAR(T,1Ø,127+I)
>2ØØ NEXT I
>21Ø GOTO 21Ø
>RUN
```

This is just the way you'll see it on the screen. You'll notice we've gotten onto the sixth line in both cases. Line 13Ø and the program from

15Ø-2ØØ demonstrate that the information is valid and properly handled by the computer.

To key in a line like 14Ø, start entering it as usual. When you get to the end of the fourth line and you enter the first "7" you'll come to a halt. Hit the ENTER key. Now type 14Ø and a FCTN down (or EDIT 14Ø) and use the right arrow to move to the end. This will enable you to move onto the next line (5th). Enter the rest of the seven's and as many eights as you can get (you'll get 12). Again, hit the ENTER key. Use the EDIT feature to get to the end again and you'll be able to move to the sixth line. Experiment with this little feature and a time will come up (as you'll see in Chapter 11) where that one extra line (or even character) is very important.

Since this isn't an approved method of creating a line, and the computer doesn't want you to be able to enter anything longer than four lines, be sure you test your program fully before you assume it's operating correctly.

```
100 REM  * BUILDING BLOCKS *
110 REM  BY T CASTLE
120 REM  AMLIST V-PB132KB
130 CALL CLEAR
140 CALL SCREEN(8)
150 DIM SET(9,30)
160 DATA 9,13,10,13,11,11,12
,11,13,9,14,9,15,5,16,5,2,15
170 FOR I=1 TO 9
180 READ A,B
190 CALL COLOR(A,B,16)
200 NEXT I
210 DATA 030F3F3F7F7FFFFF,FF
FF7F7F3F3F0F03,C0F0FCFCFEFEF
FFF,FFFFFEFEFCFCF0C0,0001071
F1F3F3F7F,7F3F3F1F1F070100
220 DATA 0080E0F8F8FCFCFE,FE
FCFCF8F8E08000,FFFFFFFFFFFFF
FFF,18183C3C7E7EFFFF,0101030
307070F0F,1F1F3F3F7F7FFFFF
230 DATA 8080C0C0E0E0F0F0,F8
F8FCFCFEFEFFFF,3C7EFFFFFFFF7
E3C

240 FOR I=95 TO 143 STEP 16
250 RESTORE 210
260 FOR K=1 TO 15
270 READ A$
280 CALL CHAR(I+K,A$)
290 NEXT K
300 NEXT I
310 CALL CHAR(40,"FF81818181
8181FF")
320 DATA " GREEN "," YELLOW
"," RED "," BLUE "," CLEAR "
,"         ",SMALL,MEDIUM,LAR
GE,TRIANGLE,CIRCLE,SQUARE
330 DATA "2103COLOR ","2103S
HAPE ","2103ROW    ","2103COL
UMN"
340 READ CL$(1),CL$(2),CL$(3
),CL$(4),CL$(5),CL$(6),SZ$(0
),SZ$(1),SZ$(2),SH$(0),SH$(1
),SH$(2)
350 READ ASK$(1),ASK$(2),ASK
$(3),ASK$(4)
360 DATA 2,2,0,0,1,1,-1,0,1,
3,-1,2,1,4,0,2,1,0,-2,1,1,-1
,0,1,1,-1,-1,1,1,99,1,2,0,0,
1
370 DATA 1,-1,0,1,3,-1,1,1,4
,0,1,1,99,0,0,0,0,1,99,2,-4,
0,0,1,-5,-2,0,1,-3,-2,2,1,-2
,0,2,1
380 DATA -1,-1,0,3,-1,-2,1,1
,-1,0,1,1,99,1,-8,0,0,1,-9,-
1,0,1,-7,-1,1,1,-6,0,1,1,99,
0,5,0,0,1
390 DATA 99,2,-1,0,0,3,-1,-1
,0,3,-1,-2,0,3,99,1,-1,0,0,2
,-1,-1,0,2,99,0,-1,0,0,1,99
400 FOR I=1 TO 9
410 IF I>3 THEN 430
420 CALL HCHAR(I,3,40,29)
430 FOR K=1 TO 30
440 READ ST
450 SET(I,K)=ST
460 IF ST=99 THEN 480
470 NEXT K
480 NEXT I
490 DATA 3,5,121,3,9,105,3,1
2,153,3,14,137,3,18,121,3,21
,153,3,23,105,3,27,137
500 DATA 3,30,121,8,4,105,8,
7,121,12,4,137,12,7,153
510 FOR G=1 TO 13
520 IF G<10 THEN 550
530 L=8
540 GOTO 560
550 L=G
560 READ R1,C1,CC
570 GOSUB 1280
580 NEXT G
590 DATA 0404A   B  C D   E
 F G   H  I,0618COLUMN,0712A
BCDEFGHIJKLMNOPQRS,0903A  B,
1303C  D,1506NEW,1703E  F
600 DATA 2003ENTER,1410R,151
0O,1610W,1503((,1603((
610 FOR G=1 TO 13
620 READ MSG$
630 GOSUB 1330
```

```
640 NEXT G
650 FOR I=8 TO 23
660 CALL HCHAR(I,12,I+57)
670 CALL HCHAR(I,13,40,19)
680 NEXT I
690 FOR AS=1 TO 4
700 MSG$=ASK$(AS)
710 GOSUB 1330
720 SND=0
730 CALL HCHAR(22,5,32,5)
740 CALL SOUND(-5,1175,0)
750 CALL HCHAR(22,4,63)
760 CALL KEY(3,ASK(AS),STAT)
770 SND=SND+1
780 IF STAT>0 THEN 810
790 CALL HCHAR(22,4,32)
800 IF SND=7 THEN 720 ELSE 7
60
810 ON AS GOTO 820,910,990,1
030
820 CALL HCHAR(24,9,32,24)
830 CALL HCHAR(22,7,ASK(AS))
840 IF ASK(AS)=70 THEN 650
850 IF (ASK(AS)>64)*(ASK(AS)
<70)THEN 860 ELSE 720
860 MSG$="2411"&CL$(ASK(AS)-
64)
870 CLR$=CL$(ASK(AS)-64)
880 GOSUB 1330
890 A1=ASK(AS)
900 GOTO 1060
910 CALL HCHAR(22,7,ASK(AS))
920 IF (ASK(AS)>64)*(ASK(AS)
<74)THEN 930 ELSE 720
930 LM=SET(ASK(AS)-64,1)
940 LX=INT((3*(ASK(AS)-64))*
.1)
950 MSG$="2409"&SZ$(LM)&CLR$
&SH$(LX)
960 GOSUB 1330
970 A2=ASK(AS)
980 GOTO 1060
990 CALL HCHAR(22,7,ASK(AS))
1000 IF (ASK(AS)>64+LM)*(ASK
(AS)<81)THEN 1010 ELSE 720
1010 A3=ASK(AS)
1020 GOTO 1060
1030 CALL HCHAR(22,7,ASK(AS)
)
1040 IF (ASK(AS)>64)*(ASK(AS
)<84-LM)THEN 1050 ELSE 720
1050 A4=ASK(AS)
1060 NEXT AS
1070 CALL SOUND(15,1319,1)
1080 CALL SOUND(15,1109,1)
1090 CALL SOUND(15,1319,1)
1100 CALL SOUND(15,1109,1)
1110 CALL SOUND(15,1319,1)
1120 CC=((A1-65)*16)+105
1130 IF CC<>169 THEN 1230
1140 J=SET(A2-64,1)+1
1150 CC=0
1160 R1=A3-57
1170 C1=A4-52
1180 ON J GOTO 1210,1200,119
0
1190 CALL HCHAR(R1-2,C1,40,J
)
1200 CALL HCHAR(R1-1,C1,40,J
)
1210 CALL HCHAR(R1,C1,40,J)
1220 GOTO 690
1230 R1=A3-57
1240 C1=A4-52
1250 L=A2-64
1260 GOSUB 1280
1270 GOTO 690
1280 FOR K=2 TO 30 STEP 4
1290 IF SET(L,K)=99 THEN 132
0
1300 CALL HCHAR(R1+SET(L,K+1
),C1+SET(L,K+2),CC+SET(L,K),
SET(L,K+3))
1310 NEXT K
1320 RETURN
1330 R1=VAL(SEG$(MSG$,1,2))
1340 C1=VAL(SEG$(MSG$,3,2))
1350 FOR I=1 TO LEN(MSG$)-4
1360 CALL HCHAR(R1,C1+I,ASC(
SEG$(MSG$,I+4,1)))
1370 NEXT I
1380 RETURN
```

HAPPY COMPUTING!

```
***********************
* 3D  TIC-TAC-TOC-TOE *
*      V-PM731KB      *
*     BY T CASTLE     *
***********************
```

DESCRIPTION. This game is similar to TIC-TAC-TOE, in that you attempt to get your "X's" or "Ø's" in any straight line before your opponent. This game is a good bit more challenging since you must get 4 marks in a row and you're working in three dimensions. The screen display looks very similar to the perspective shot we used as an example in Chapter 7. The display has a light yellow background and each horizontal "plane" has 16 colored blocks connected by fine black lines. From top to bottom the blocks are colored green, white, red, and blue. At the bottom, the question appears "X - ENTER I,J,K? ". The question alternates between "X" and "Ø" input prompts. The user enters 3 numbers, each between 1 and 4. The computer will automatically add commas between the numbers you enter. If you have entered 1 or 2 numbers and you decide you have made a mistake, a Function 3 will erase your entry and permit you to reenter. Only appropriate responses are permitted. After each entry, the computer will go through a series of "beeps" while it checks for a possible "WIN" situtation. If it finds one, the computer will indicate the winner and you can play a new game by hitting any key.

This game is more challenging than it looks at first glance. By our calculations, and we won't claim they're 1ØØ% correct, we find 92 possible "WIN" situations. We're not sure whether you can ever come up with a "DRAW". The subroutine beginning at line 114Ø checks 16 directions for a possible win. To the best of our knowledge this'll catch any win situation. It's not as challenging as chess but, in our opinion, it's probably a little more thought provoking than checkers.

NOTES. In a slight departure from our normal method of building a program, we've written this almost exclusively in "straight line" fashion. In fact, the only GOSUB in the program is the verification routine. It's still documented in very much the same way as it would be if it was broken into subroutines. You could actually enter this program through 112Ø, remove line 1Ø8Ø and 1Ø9Ø and the program would run just fine. You would have to be responsible for determining a win.

In order to make it possible to check for wins, we've internally built a theoretical 1Ø X 1Ø X 1Ø array, initially filled with zeros. You could think of each of the "blocks" that appear on the screen as being a 4 X 4 X 4 array, surrounded by 3 more zeros in every direction. As each entry is made, we change the zero to either a "1" if it is an "X" entry or a "6" if it is a "Ø" entry. By adding the value of all of these little cells, for 3 positions, in every direction off of the last entry, we can determine if there's a winner or not. If the total of the individual cells is 4, we know that "X" wins. If the total is 24 then "Ø" wins. Any other combination is not a winner, and no combination of Ø's and X's can come up with these figures. The calculations that put the 1's and 6's in the string array are found in lines 1Ø1Ø and 1Ø7Ø.

```
100 REM   ******************
110 REM   3D TIC-TAC-TOC-TOE
120 REM   ******************
130 REM
140 REM   BY T CASTLE
150 REM   AMLIST V-PM731KB
160 REM
170 CALL CLEAR
180 OPTION BASE 1
190 DIM T$(10,10)
200 DIM TOT(20)
210 CALL SCREEN(12)
220 DATA 12,5,13,7,14,16,15,
3,16,2
230 FOR I=1 TO 5
240 READ A,B
250 CALL COLOR(A,B,1)
260 NEXT I
270 DATA ØØØØ3E3E3E3E3EØØ,ØØ
3F3F3F3F3F3FØØ,ØØ7F7F7F7F7F7
F7F,FFFFFFFFFFFFFFFF
280 DATA ØØØØØØØØFFØØØØØØ,8Ø
4Ø2Ø1ØØ8Ø4Ø2Ø1,ØØØØØØFFFFØØØ
ØØØ,8Ø4Ø2ØFFFFØ4Ø2Ø1,8Ø4Ø2Ø1
ØFFØ4Ø2Ø1
290 FOR K=12Ø TO 144 STEP 8
300 RESTORE 27Ø
310 FOR I=K TO K+3
320 READ A$
330 CALL CHAR(I,A$)
340 NEXT I
350 NEXT K
360 FOR I=152 TO 156
370 READ A$
380 CALL CHAR(I,A$)
390 NEXT I
400 CALL CLEAR
410 FOR I=1 TO 1Ø
420 FOR J=1 TO 1Ø
430 T$(I,J)="ØØØØØØØØØØ"
440 NEXT J
450 NEXT I
460 B=8
470 CKW=Ø
480 B=8
490 FOR I=1 TO 16 STEP 5
500 CALL HCHAR(I,5,152,17)
510 CALL HCHAR(I+6,11,154,17
)
520 CALL HCHAR(I+1,5,153)
530 CALL HCHAR(I+3,7,153)
540 CALL HCHAR(I+5,9,153)
550 CALL HCHAR(I+1,23,153)
560 CALL HCHAR(I+3,25,153)
570 CALL HCHAR(I+5,27,153)
580 B=B-8
590 FOR J=4 TO 22 STEP 6
600 K=143
610 FOR L=Ø TO 6 STEP 2
620 K=K+1
630 CALL HCHAR(I+L,J+L,K+B)
640 NEXT L
650 NEXT J
660 NEXT I
670 FOR I=6 TO 16 STEP 5
680 CALL HCHAR(I,9,156)
690 CALL HCHAR(I+1,23,155)
700 NEXT I
710 REM  INPUT STATEMENTS
720 PL=1
730 CALL HCHAR(24,1,32,32)
740 REM  MESSAGE INPUT
750 IF PL=1 THEN 78Ø
760 MSG$="Ø -ENTER I,J,K? "
770 GOTO 79Ø
780 MSG$="X -ENTER I,J,K? "
790 FOR I=1 TO LEN(MSG$)
800 CALL HCHAR(24,1+I,ASC(SE
G$(MSG$,I,1)))
810 NEXT I
820 FOR I=1 TO 3
830 CALL KEY(3,KY,ST)
840 IF ST<1 THEN 83Ø
850 IF KY=7 THEN 91Ø
860 IF (KY<49)+(KY>52)THEN 8
3Ø
870 CALL HCHAR(24,2Ø+(I*2),K
Y)
880 IF I=3 THEN 9ØØ
890 CALL HCHAR(24,21+(I*2),4
4)
900 M(I)=VAL(CHR$(KY))
910 NEXT I
920 IF KY=7 THEN 73Ø
930 REM   MAKES MOVE
940 ROW=(((M(1)*4)-(4-M(1)))
+6)-((M(3)*2)-2)
950 COL=((((M(2)*3)-1)*2)+6)
-((M(3)*2)-2)
960 IF PL=1 THEN 1Ø3Ø
```

```
970 MARK$="Ø"
980 CALL GCHAR(ROW,COL,NR)
990 IF (NR=48)+(NR=88)THEN 7
3Ø
1ØØØ CALL HCHAR(ROW,COL,ASC(
MARK$))
1Ø1Ø T$(M(1)+3,M(2)+3)=SEG$(
T$(M(1)+3,M(2)+3),1,M(3)+2)&
"6"&SEG$(T$(M(1)+3,M(2)+3),M
(3)+4,11-(M(3)+4))
1Ø2Ø GOTO 1Ø8Ø
1Ø3Ø MARK$="X"
1Ø4Ø CALL GCHAR(ROW,COL,NR)
1Ø5Ø IF (NR=48)+(NR=88)THEN
73Ø
1Ø6Ø CALL HCHAR(ROW,COL,ASC(
MARK$))
1Ø7Ø T$(M(1)+3,M(2)+3)=SEG$(
T$(M(1)+3,M(2)+3),1,M(3)+2)&
"1"&SEG$(T$(M(1)+3,M(2)+3),M
(3)+4,11-(M(3)+4))
1Ø8Ø GOSUB 113Ø
1Ø9Ø IF CKW=1 THEN 4ØØ
11ØØ IF PL=1 THEN 111Ø ELSE
72Ø
111Ø PL=Ø
112Ø GOTO 73Ø
113Ø REM  VERIFY FOR WIN
114Ø A=M(1)+3
115Ø B=M(2)+3
116Ø C=M(3)+3
117Ø J=4
118Ø FOR I=-3 TO +3
119Ø J=J-1
12ØØ TOT(1)=TOT(1)+VAL(SEG$(
T$(A+I,B),C,1))
121Ø TOT(2)=TOT(2)+VAL(SEG$(
T$(A,B+I),C,1))
122Ø TOT(3)=TOT(3)+VAL(SEG$(
T$(A,B),C+I,1))
123Ø TOT(4)=TOT(4)+VAL(SEG$(
T$(A+I,B+I),C,1))
124Ø CALL SOUND(1Ø,12ØØ,1)
125Ø CALL HCHAR(ROW,COL,32)
126Ø CALL SOUND(1Ø,12ØØ,1)
127Ø TOT(5)=TOT(5)+VAL(SEG$(
T$(A+I,B),C+I,1))
128Ø TOT(6)=TOT(6)+VAL(SEG$(
T$(A,B+I),C+I,1))
129Ø TOT(7)=TOT(7)+VAL(SEG$(
T$(A+I,B+I),C+I,1))

13ØØ TOT(8)=TOT(8)+VAL(SEG$(
T$(A+I,B+J),C,1))
131Ø TOT(9)=TOT(9)+VAL(SEG$(
T$(A+J,B+I),C,1))
132Ø TOT(1Ø)=TOT(1Ø)+VAL(SEG
$(T$(A+I,B),C+J,1))
133Ø TOT(11)=TOT(11)+VAL(SEG
$(T$(A+J,B),C+I,1))
134Ø TOT(12)=TOT(12)+VAL(SEG
$(T$(A,B+J),C+I,1))
135Ø CALL SOUND(1Ø,12ØØ,1)
136Ø CALL HCHAR(ROW,COL,ASC(
MARK$))
137Ø CALL SOUND(1Ø,12ØØ,1)
138Ø TOT(13)=TOT(13)+VAL(SEG
$(T$(A,B+I),C+J,1))
139Ø TOT(14)=TOT(14)+VAL(SEG
$(T$(A+I,B+J),C+J,1))
14ØØ TOT(15)=TOT(15)+VAL(SEG
$(T$(A+J,B+I),C+J,1))
141Ø TOT(16)=TOT(16)+VAL(SEG
$(T$(A+J,B+J),C+I,1))
142Ø NEXT I
143Ø FOR I=1 TO 16
144Ø IF (TOT(I)=4)+(TOT(I)=2
4)THEN 145Ø ELSE 154Ø
145Ø CALL HCHAR(24,1,32,32)
146Ø MSG$=MARK$&" - WINS!
    HIT ANY KEY!"
147Ø FOR J=1 TO LEN(MSG$)
148Ø CALL HCHAR(24,1+J,ASC(S
EG$(MSG$,J,1)))
149Ø CALL SOUND(5,1ØØØ+(J*5Ø
),1)
15ØØ NEXT J
151Ø CKW=1
152Ø CALL KEY(3,KY,ST)
153Ø IF ST=Ø THEN 152Ø
154Ø TOT(I)=Ø
155Ø NEXT I
156Ø RETURN
```

HAPPY COMPUTING!

CHAPTER ELEVEN

Algorithms

GENERAL. You've probably heard it said that mathematicians make good programmers. You may have also encountered some calculations in our programs, and programs from other sources, that have "baffled" you in their construction and what they do. These calculations are sometimes referred to as "algorithms" or formulas. Properly constructed, these can be extremely powerful tools, particularly in combination with the DEF statement, and can often be used in place of entire 5 or 10 line subroutines. Even more than some of the techniques offered in the previous chapter, they offer the potential for real progress in condensing and refining programs. In this chapter we're going to give you several new algorithms and we're going to discuss some that have been used to this point; further, we hope to take some of the mystery out of them so that you can create your own. For those of you who feel that you're not "well grounded" in math, "have no fear", most of the "experts" didn't just write a 100 character formula from scratch either. Many experienced programmers spend as much time developing these little gems as they do on the rest of the program. Of course, the hours spent the first time are saved in every program that requires their use thereafter. If you use a little logic, imagination, and a step-by-step approach you can create these yourself.

Right/Left Justification. By now all of you have become familiar with the idea of "padding" string data so that it sits either to the right or left hand side of a "field". The following takes a five digit number (converted to a string) and places it to the right in a field of 10 characters.

```
>100 INPUT A
>110 A$=STR$(A)
>120 IF LEN(A$)=10 THEN 150
>130 A$=" "&A$
>140 GOTO 120
>150 PRINT A$
>160 END
```

The above works very efficiently, but to get from A to A$, right justified, consumes 5 lines of code. The following does exactly the same thing in just one line.

```
>100 A=23.45
>120 PRINT SEG$("          "&
 STR$(A),LEN(STR$(A))+1,10)
>130 END
```

If you're working on a program with a lot of data, you might need to do this dozens of times. You can put the first example in a subroutine, but you still have to send it to the subroutine each time and RETURN. Isn't it much easier to perform the entire operation on a single line? Better yet, to keep from typing it after each input, put it in a DEF statement.

```
>100 DEF QR$=SEG$("
 "&STR$(Q1),LEN(STR$(Q1))+1,1
 0)
>110 REM PROGRAM HERE
>500 INPUT "1ST INPUT   ":Q1
>510 A$=QR$
```

```
>52Ø INPUT "2ND INPUT   ":Q1
>53Ø B$=QR$
>54Ø PRINT A$
>55Ø PRINT B$
>56Ø END
```

Using the DEF statement, anytime you ask for QR$, either for a PRINT statement or to set it equal to some other variable, QR$ will be based on the current value of Q1. Now we're going to develop this whole thing a lot further, so before it becomes to confusing, let's go back to the first method of justification and see how it was converted to one line.

To develop an algorithm you have to consider what information you have to work with and what you want out of it. If you want to do something in one line, you can forget about FOR-NEXT loops or anything that involves counting or incrementing a value (like we do in our first example). Using nothing but the variable itself in the equation, what do we know about a number that is inputted? If the number is called A, we know: as a string it's STR$(A); its integer value is INT(A); absolute value is ABS(A); and the length of each of these could also be determined as LEN(STR$(A)), LEN(STR$(INT(A))), LEN(STR$(ABS(A))). Of all of this information, nothing will give us a string 1Ø characters long, and that's what we need. If it was more than 1Ø characters long, we could easily get 1Ø out of it by using the SEG$ command. All we would have to do is specify the variable such as STR$(A), a starting position, and the repetition factor of 1Ø. In a formula this would be:

```
>1ØØ A$=SEG$(STR$(A),Start Po
 int,1Ø)
```

We can make STR$(A) equal to or more than 1Ø characters long by simply adding 1Ø blanks. This would give us:

```
>1ØØ A$=SEG$("          "&STR
 $(A),Start Point,1Ø)
```

All we lack now is a calculation for starting point. The easiest way to find something like this is to mentally try a few numbers. If (A) was equal to 52.3, the string that we're trying to get 1Ø characters out of would be 14 long, or the length of the string value of (A), which is 4, plus the 1Ø we added. We know that they're sitting to the right hand side of the string, so we need the characters from 5 through 14. Remember the value of 5. If you use a number "4" for the value of (A), the same calculation would mean you start at position 2. If the number was 729.3 you would start at 6. Notice, in each case you start at a number one higher than the length of the STR$(A). So our final formula becomes:

```
>11Ø A$=SEG$("          "&STR
 $(A),LEN(STR$(A))+1,1Ø)
```

If you're working in dollars and cents, and you perform a division calculation on the value of (A) before it reaches your command that will right justify, you might get something like "   45.2384". Sometimes you'll need to round the number prior to getting it justified. To figure our algorithm, let's do it the long way first.

```
>1ØØ A=45.2384
>11Ø A=INT(A*1ØØ+.5)/1ØØ
>12Ø A$=SEG$("          "&STR
 $(A),LEN(STR$(A))+1,1Ø)
>13Ø PRINT A$
```

It's just as easy to remove line 11Ø and replace the A's in line 12Ø with the rounding formula. The result is:

```
>1ØØ A=45.2384
>12Ø A$=SEG$("          "&STR
 $(INT(A*1ØØ+.5)/1ØØ),LEN(STR
 $(INT(A*1ØØ+.5)/1ØØ))+1,1Ø)
>13Ø PRINT A$
```

All you have to do is be careful, do one thing at a time, test it, and make sure you keep all of your parenthesis straight. As a "one liner" this one is very handy, but you would still get dollars and cents columns that look like this:

```
  54.2
 29.85
    22
```

The following routine, which involves the use of four DEF statements will handle any number from -999,999.99 to 9,999,999.99 (commas added for clarity). It will round, right justify, and add the necessary trailing zeros. If you use it just as written, you need an input value of Q and you'll receive the rounded value as (A) and the right justified string as A$.

```
>1ØØ DEF NR1$=STR$(INT(VAL(ST
 R$((INT(Q*1ØØ+.5))/1ØØ))))
>11Ø DEF NR2$=SEG$(STR$(1ØØ+(
 1ØØ*((INT(Q*1ØØ+.5)/1ØØ)-INT
 (INT(Q*1ØØ+.5)/1ØØ)))),2,2)
>12Ø DEF NR3$=SEG$("
  "&NR1$&"."&NR2$,LEN(NR1$)+L
 EN(NR2$)+2,1Ø)
>13Ø DEF NR4$=SEG$("
 -"&NR1$&"."&NR2$,LEN(NR1$)+L
 EN(NR2$)+2,1Ø)
>14Ø INPUT Q
>15Ø IF Q>=Ø THEN 21Ø
>16Ø Q=ABS(Q)
>17Ø A=VAL(NR4$)
>18Ø A$=NR4$
>19Ø PRINT A$;"   ";A
>2ØØ GOTO 14Ø
>21Ø A=VAL(NR3$)
>22Ø A$=VAL(NR3$)
>23Ø PRINT A$;"   ";A
>24Ø GOTO 14Ø
```

Handling negative numbers does present a problem, because if you ask for an INTeger value of any negative number, the normal rules for rounding do not apply. For instance, if you ask for INT(-1.3) you get -2 as an answer. To make sure we got true rounding we had to first treat the number as an ABSolute value (line 16Ø) and then round and justify. If there is no possibility of a negative number in your calculation you can remove line 13Ø, and 15Ø-2ØØ.

```
>1ØØ DEF NR1$=STR$(INT(VAL(ST
 R$((INT(Q*1ØØ+.5))/1ØØ))))
>11Ø DEF NR2$=SEG$(STR$(1ØØ+(
 1ØØ*((INT(Q*1ØØ+.5)/1ØØ)-INT
 (INT(Q*1ØØ+.5)/1ØØ)))),2,2)
>12Ø DEF NR3$=SEG$("
  "&NR1$&"."&NR2$,LEN(NR1$)+L
 EN(NR2$)+2,1Ø)
>14Ø INPUT Q
>21Ø A=VAL(NR3$)
>22Ø A$=VAL(NR3$)
>23Ø PRINT A$;"   ";A
>24Ø GOTO 14Ø
```

This program really isn't as complex as it seems. Using 34.567 as an example: line 1ØØ rounds it to 34.57 and creates a string value equal to the integer value of that (i.e. "34"). Line 11Ø extracts the last three digits from the rounded value (i.e. .57), multiplies them times 1ØØ and adds 1ØØ (now it equals 157), and creates a string value equal to the last two digits of that number (i.e. "57"). If you don't think the multiplication and addition of 1ØØ are

necessary, try it on a zero or whole number without it and see what you get. Lastly, in line 12Ø we add the two strings together with a "." between them. The only difference in 13Ø is that we also add a minus sign in front of it.

We have several other algorithms to cover, so we don't want to spend too much more time on this one. To left justify you would just add your spaces (such as found in 13Ø) after the NR1$&"."&NR2$. Your starting point would always be 1 and you would still take 1Ø repetitions. To right or left justify alpha information, such as names, would be far easier since you would only use the inputted string and no calculations would be required. You can find an example of this in line 82Ø or 94Ø of "Memory Jogger". To keep from having to key in "spaces" each time we wanted to justify, we created a string called AD$ at the beginning of the program that contained 15 spaces. This gave us more than enough to use at any point in the program.

Calender vs Ordinal Dates. In a previous chapter we briefly mentioned the idea of putting the year in front of the rest of the date when storing this type of information in data files or memory. Actually, unless you're absolutely sure that you'll never want to know an interval between two dates, the most efficient way to store them is in their "Ordinal" form. Under this system, a date is its position within the year: for instance, 1/21/83 is number 21 (21st day) and 12/31/83 is number 365. To determine the interval, all you need to do is subtract 21 from 365. If you're working from year to year, place the year in front of the number, i.e. 82Ø21 (1/21/82), and 83365 (12/31/83). An adjustment will be necessary here, but the earlier year is still the lowest number. This method of ordinal dating is probably sufficient for most modern business practices; however, be aware that it's not 1ØØ% accurate since it does not take into account leap years. To get the true deviation between dates you may want to consider reading up on the Julian method. This system takes into account all leap years and is, in effect, a consecutive numbering system which begins with day zero on November 24, -4713. Don't try to set up algorithms based on this date alone. There are some adjustments that took place in 1582 which could throw you off. Check the library for specific details on the differences between the various calendars.

As we said, except for the fact that it does not make the adjustment for leap years, this method is precise to the day. Using DEF statements, the following program converts normal dates to a five digit code, converts the code to a normal date, and determines intervals between dates. We need to mention that line 15Ø of this program requires the addition of one parenthesis on the fifth line. If you have trouble entering this, go back and read the information on "Adding Data Lines" in Chapter 1Ø.

```
>1ØØ DIM AJ(12),AI(12)
>11Ø DATA Ø,-2,1,Ø,-1,-1,Ø,-1
 ,Ø,-2,1,-2,1,-3,2,-4,3,-4,3,
 -5,4,-5,4,-6
>12Ø FOR I=1 TO 12
>13Ø READ AJ(I),AI(I)
>14Ø NEXT I
>15Ø DEF DT1$=STR$(((VAL(SEG$
 (Q$,1,2))*3Ø)-3Ø)+(VAL(SEG$(
 Q$,3,2)))+(VAL(SEG$(Q$,5,2))
 *1ØØØ)+AJ(VAL(SEG$(Q$,1,2)))
 )
```

```
>160 DEF HL=(INT((VAL(SEG$(Q1
 $,3,3))+AI(INT(VAL(SEG$(Q1$,
 3,3))/30)))/30)+1)+100
>170 DEF HM=(VAL(SEG$(Q1$,3,3
 )))-(AJ(HL-100))-(((HL-100)-
 1)*30)+100
>180 DEF DT2$=SEG$(STR$(HL),2
 ,2)&SEG$(STR$(HM),2,2)&SEG$(
 Q1$,1,2)
>190 DEF DV=Q1-Q2-(((VAL(SEG$
 (STR$(Q1),1,2)))-(VAL(SEG$(S
 TR$(Q2),1,2))))*635)
>200 CALL CLEAR
>210 INPUT "DATE-031283 ":Q$
>220 PRINT "CODE IS       ";DT
 1$::
>230 INPUT "ENTER CODE  ":Q1$
>240 PRINT "DATE IS       ";DT2
 $::
>250 INPUT "NEW-OLD, I.E.
            012383,102043 ":N
 1$,N2$
>260 Q$=N1$
>270 Q1=VAL(DT1$)
>280 Q$=N2$
>290 Q2=VAL(DT1$)
>300 PRINT "DEVIATION IS  ";D
 V::
>310 CALL KEY(3,KY,S)
>320 IF S=0 THEN 310 ELSE 200
>RUN
```

To create these algorithms we started by trying to convert a date to an ordinal code. We began by inputting a date and then we took off the figures representing the year, since the ordinal date is based only on day and month within a year.

```
>100 Q$="031283"
>110 YR$=SEG$(Q$,5,2)
>120 Q$=SEG$(Q$,1,4)
```

If all months were 30 days, or some other constant figure, and we took the month figure times 30 we would get the last day of each month, i.e. 01 (JAN) would be 30, 02 (FEB) would be 60, etc. By subtracting 30 from that figure we would get to the last day of the previous month. If we just added the day of the month to the amount of days elapsed prior to that month we would have our number.

```
>130 M=(VAL(SEG$(Q$,1,2))*30)
 -30
>140 D=VAL(SEG$(Q$,3,2))
>150 DATE=M+D
>160 PRINT DATE
>RUN
```

Unfortunately, all months are not equal and, although we were in the ball park, the answer wasn't quite right. At this point we created a little ARRAY called AJ(n) and filled it with zeros. By trial and error we were able to figure out the monthly adjustment necessary to the DATE figure above to make it correct. We simply used the value of the first two digits of Q$ (the month) as a subscript to indicate which adjustment to use.

```
>75 DIM AJ(12)
>80 DATA 0,1,-1,0,0,1,1,2,3,3
 ,4,4
>85 FOR I=1 TO 12
>90 READ AJ(I)
>95 NEXT I
>100 INPUT Q$
>110 YR$=SEG$(Q$,5,2)
>120 Q$=SEG$(Q$,1,4)
>130 M=(VAL(SEG$(Q$,1,2))*30)
 -30
>140 D=VAL(SEG$(Q$,3,2))
>150 DATE=M+D+AJ(VAL(SEG$(Q$,
 1,2)))
>160 PRINT DATE
>170 GOTO 100
>RUN
```

Once the details were worked out, we just combined it all into the statement in line 150. In order to

add the year figure of "83", we multiplied it times 1000 and then added the ordinal date. When we had the raw figure, we put parenthesis around the whole thing and added a STR$ command.

As a separate program, we did the same thing starting with an ordinal date and worked backwards to get the calender date. It took a separate set of adjustments and the statements got pretty long so we had to use more than one DEF statement. When we had it done we combined it with the previous program. Using these two variables we added one more DEF statement to calculate the difference between two dates.

Use your imagination and you may be able to come up with a lot of nice uses for these routines. Have you ever seen a table that can tell you what day of the week it was on any given date? You'll need some additional adjustments for this such as leap year (the year is always evenly divisible by 4). If you're really going to go a long way back, bear in mind that only centuries (like, 1600, 2000, 2400, etc.) which are divisible by 400 are actually leap years, even though all centuries are divisible by 4. If you take these factors into consideration and you know that each week is seven days you can build your own calender for any point in time. Since the interval indicates the passing of time, astrology buffs may find it handy for calculating the future or past positions of stars and constellations.

A really heavy math background isn't really required to create these types of formulas, just patience. Work out each program in small increments, using very simple addition, subtraction, multiplication and division. Look for relationships between numbers and don't be afraid to use an "adjustment" if necessary. If you can't solve a part of the puzzle, move on to the next step anyway. Many times, in working on a later part, you'll discover an answer to the first part. Just keep moving forward. When the "theory" of the calculation is proven, then you can CONDENSE the statements into one or more long statements.

Creating Coding Systems. The algorithm we're about to describe has tremendous potential if you need to hold a lot of information in memory or if you want to transfer a lot of data to and from a storage device. The Golf Handicap program simply would not have been possible had it not been for this type of system. Even if you're not a golfer, bear with this discussion and we'll show you some other uses.

To keep a handicap on just one person you must know what his score was on the last 20 rounds of golf. For most this is normally a number between 70 and 110. A true handicap isn't based just on this score; rather, it's a factor of the course on which it is played. In golf, courses are rated using a number such as 71.3, 72.5, etc. To keep records for 12 golfers would require 12 X 20 scores (240), and two sets of numbers for each (480 total numbers). In addition, some golfers want to keep hole by hole information on their rounds. In bowling we only have 3 scores to worry about - in golf there are 18 holes and a par value for each. For just eight individual rounds that would add another 288 numbers. Adding those together at 8 bytes a piece, plus names, course ratings, and other base information, we would have just about

consumed memory even if we didn't have a program. Further, you could almost play a round of golf while it loaded the previous data. Even our previous method of using string arrays and packing the information on a data line didn't accomplish the job. Building on this idea and using a specialized "shorthand code" similar to the way it's used on graphics, we were able to cram all this information on just seven 192 character data lines (just a little over 1300 bytes). Here's how it was done.

In traditional numbering systems there are only 10 characters to be used in any one position of a number (the numbers 0-9). In Hexidecimal (for graphics) they use 0-9 and the letters A-F. This means 16 characters can appear in any one position. On the standard character code chart there are at least 96 different characters (32-127) if you don't count the special sets. We saw no reason why we couldn't use these in place of numbers. To decide how and what we needed, we looked at the data required for one round, such as a score of 83 on a course with a rating of 72.5. Putting these side by side we had 8372.5. We figured if we did it carefully, using the SEG$ command we could eliminate the need for the decimal and just store 83725. Next we started to figure what we could do with a coding system for reducing it further. We tried several possibilities and finally settled on using the letter A to represent zero, and each of the other characters from 66 to (and including) 124 to represent the numbers 1 to 59. Since a golfers' score might go to 110 or 120 we needed to code a number up to about 120729 (six digits, and something over about 120 thousand) and we wanted to do it with just 3 characters. Using the code AAA as our starting point, we established "zero". A number "one" would be AAB, and the number 59 would be AA|. If you don't recognize it, this symbol "|" is a FCTN-A on your keyboard. Since the computer knows the character value, we could arrive at the number by just subtracting 65. When we got to sixty we just started incrementing the second character. This meant that 60 was ABA, and 3599 was A||. To go to 3600 we started working on the first character (3600=BAA.) If you could go 60 groupings like this you could go as high as 215999 with 3 letters. That was more than enough for our purposes.

The algorithms followed the logic above and were really easier than either of the previous ones discussed. They are found in lines 290 and 300 of the handicap program and repeated here:

```
>100 DEF R$=CHR$((INT(E/3600)
 )+65)&CHR$(INT((E-((INT(E/36
 00)))*3600)/60)+65)&CHR$(INT
 (E-((INT(E/60)))*60)+65)
>110 DEF R=((ASC(SEG$(E$,1,1)
 )-65)*3600)+((ASC(SEG$(E$,2,
 1))-65)*60)+((ASC(SEG$(E$,3,
 1))-65))
>120 INPUT "ENTER NR.   ":E
>130 PRINT "CODE    ";R$::
>140 INPUT "ENTER CODE  ":E$
>150 PRINT "NUMBER ";R::
>160 GOTO 120
>RUN
```

The above uses E as an input and returns the code as R$. To reverse the process just enter the code as E$ and the number is returned as R. In a working program, be sure you immediately set the R or R$ equal to some other variable since its value may change from time to time if E or E$ is used elsewhere in the program.

Now here's a slightly different version which will enable you to create your own personal coding system. Instead of putting in a fixed value for starting point (i.e. 65 above) and for the spread (6Ø above), you can INPUT your own. In characters that you can read on the screen, you can work with 33 through 126, or 94 total characters. Where it says START, enter 33. When it asks for SPREAD enter 94. Try any number between Ø and 83Ø,583. It will be represented by a simple 3 digit code.

```
>1ØØ INPUT "START    ":J
>11Ø INPUT "SPREAD   ":J1
>12Ø DEF R$=CHR$((INT(E/J1^2)
 )+J)&CHR$(INT((E-((INT(E/J1^
 2)))*J1^2)/J1)+J)&CHR$(INT(E
 -((INT(E/J1)))*J1)+J)
>13Ø DEF R=((ASC(SEG$(E$,1,1)
 )-J)*J1^2)+((ASC(SEG$(E$,2,1
 ))-J)*J1)+((ASC(SEG$(E$,3,1)
 )-J))
>14Ø INPUT "ENTER NR.   ":E
>15Ø PRINT "CODE    ";R$::
>16Ø E$=R$
>17Ø PRINT "NUMBER ";R::
>18Ø GOTO 14Ø
>RUN
```

To determine how many numbers you can represent with 1, 2, 3 or 4 digit codes, use the number of digits in the code as your exponential figure and take your spread to that power. The highest number will be that figure, minus 1. For instance, a spread of 94 with a three digit code will take any number to (94^3)-1. Tell your computer to print that and you'll get 83Ø,583.

Now that you know about it, what can you do with it? Let's look at telephone numbers like (4Ø4) 292-Ø576. If you had lot of these to store in a data file you could eliminate all of the miscellaneous punctuation and store it as 4Ø4292Ø576. As a 1Ø digit number, this is most efficiently stored as a numeric variable which consumes 8 bytes. As a string it would consume 1Ø bytes. If you broke it in the middle so that you had two numbers (4Ø429 and 2Ø576) you could also store it as two 3 digit codes which would only require 6 bytes. A spread of just 47 with a three digit code will handle any 5 digit ZIP code. In the Baseball Stats program we kept track of 9, single digit (1-9) statistics on each boy for each game. We did this with one string of data, i.e. "Ø1ØØ3Ø812". By modifying our DEF statement to a 5 digit code we could have taken care of all 9 stats. Incidently, if you want to sort this pseudo numeric data, there's no necessity to convert it to a numeric variable before sorting. Sort the code as a string and it will be in numeric order when converted.

Should You Condense Your Program? In the last two chapters we've shown you a number of ways to refine your programs, cram more data into memory, and reduce lines of code. The Building Blocks program is proof that substantial reductions can be made; however, we need to mention, in fairness, that the process is quite time consuming. What you need to ask yourself is "for a program to be useful, is it necessary, or even desirable, to condense and refine it to this degree?" We mentioned at the beginning of Chapter 1Ø that you need to start thinking about condensing even before you start writing a program; the process continues throughout programming; and may or may not continue afterwards. Let's analyze what we meant.

If you know before you start that the program is going to require a lot of data, sorting, character definitions, etc., by all means think it out thoroughly before you begin. Your choice of using codes, arrays, etc. could make the difference between success and failure. If that's not the case, you're better off to just "jump in" and write it using the techniques that first come to mind, with the object being to "get it running" from beginning to end. If you want to use a lot of IF statements, instead of an ON ___ GOTO, feel free to do so. One works as efficiently as the other in actual practice. If you're on a subroutine that you've used many times before, and you already know the shortcuts, by all means use them, but don't spend time searching for them. When it's finished, and all of the print statements work properly, data files load and save, menus are operable, etc., then you can decide whether to condense it or not.

The short, but seemingly powerful, programs you pick up from time to time were refined specifically to eliminate unnecessary lines. There are relatively few programmers who can write a totally condensed program from beginning to end. In the majority of cases, those programs started out as 300 or 400 line programs and, after the direction of the program was clear, they were rewritten into their shorter versions. The only one who truly benefits from a highly condensed and refined program is the next person who must key it into the computer. They benefit because they have fewer lines to enter; however, there is a price to pay. A highly condensed program is often very difficult to debug. Further, if they want to make modifications (adding extra names, deleting sections, etc.) they may find it all interconnected in a way that makes it almost impossible. The object at that point is to have "a place for everything and everything in its place". This often means little room for adding something "in between" and the inability to pull anything out without the whole structure crumbling. Unless you're preparing programs for publication by a magazine, the benefits, compared to the effort, probably aren't worthwhile. If it's your own program, for your own family use, no one else may ever have to enter it again. If it operates with 500 or 600 lines of code, there's no reason it can't remain that way forever.

If all of the programs in this series were condensed to the maximum degree, they would be far shorter, but also far less understandable for the novice. We resorted to more sophisticated techniques only when the situation required it. If you'd like to pursue it, simply as an exercise, take something like "Kamakaze Run", and see how many lines you can take out without changing the results.

```
********************
*   MONEY PLANNER  *
*     V-PL631KB    *
*    BY T CASTLE   *
********************
```

DESCRIPTION. This program includes four main calculations involving the present and future value of money. Following is a description of how each is used and the formulas from which our calculations were derived.

1. Future Value of Current Sum of Money. If you invested \$5,ØØØ.ØØ today at 8.5% interest, compounded quarterly, how much would you have in 5 years? Using option 1, the screen display and answers are as follows:

```
FUTURE VALUE OF CURRENT SUM

CURRENT AMT?          5ØØØ
HOW MANY MONTHS?        6Ø
ANNUAL RATE?          .Ø85

COMPOUNDED-
 1- DAILY     4- QUARTERLY
 2- WEEKLY    5- SEMI ANNUAL
 3- MONTHLY   6- ANNUAL

SELECTION?              4

FUTURE  VALUE OF
PRESENT  AMOUNT= 7613.97
```

2. Present Value of Future Sum of Money. If you could invest money today at 9% annual interest, compounded daily, how much would you have to invest to have \$1Ø,ØØØ, ten years from now? Using option 2, enter: 1ØØØØ, 12Ø, .Ø9, and 1. The answer displayed at the bottom is 4Ø66.15. Note that the time period is always entered in months and the annual percentage rate is always expressed as a decimal (i.e. 3% is .Ø3).

3. Future Value of Steady Payments - Annuity. Example 1. Suppose you wanted \$7,ØØØ.ØØ six years from now. If you could put some money into an account on a monthly basis, earning 8.75% interest, how much would you have to put in per month for six years in order to have \$7,ØØØ at then end of the six years. Using option 3, enter: 7ØØØ (Future Sum); 72 (Months); .Ø875 (Rate); and 3 (Frequency). The answer is display as 74.27 (per month).

Example 2. Suppose you wanted to start a savings account and decided to put in \$2Ø.ØØ per week. If your bank paid 6.75% interest, how much would you have in 3 years? Using option 3 again, enter: Ø (Future Sum); 2Ø (Steady Amt); 36 (Months); .Ø675 (Rate); and 2 (Frequency). Two answers are displayed. The first answer is the amount you would have if the payments (deposits) are made in "Arrears", or at the end of each weekly period; and the second answer is the value if made in "Advance", or at the beginning of each weekly period.

4. Present Value of Steady Payments - Annuity. Example 1. Suppose you just won a \$1Ø,ØØØ prize in a lottery and you could invest it at 11% annual rate of interest. How much could you take out per month, for the next 1Ø years, so that at then end of that period of time all of the 1Ø,ØØØ plus interest was gone? Using option 4, enter: 1ØØØØ (Present Sum); 12Ø (Months); .11 (Rate); and 3 (Frequency). The answer is display as 137.75 (per month).

Example 2. Using the same lottery prize, suppose you wanted to "blow" some of the money now, but still be able to draw an extra 1ØØ a month for the next five years. How much would you have to invest and how much could

you spend now? Using option 4 again, enter: 0 (Present Sum); 100 (Steady Amt); 120 (Months); .11 (Rate); and 3 (Frequency). Two answers are displayed: 7259.53 and 7326.07 The first answer is the amount you would need to invest if the payments (deposits) are made in "Arrears", or at the end of each monthly period; and the second answer is the value if made in "Advance", or at the beginning of each monthly period. Using the advance figure, you would invest $7326.07 and you could spend $2,673.93.

FORMULAS. Following are the four basic formulas used in the above calculations.

1. Future Value. $F = P(1+r)$

2. Present Value. $P = F\ (1+r)$

3. Future Value of Annuity in Arrears $F = A\left[\dfrac{(1+r) - 1}{r}\right]$

4. Future Value of Annuity in Arrears $P = A\left[\dfrac{1-(1+r)}{r}\right]$

In each of the above formulas: F=Future Value; P=Principal Amount; r=rate of interest; n=periods; and A=Annuity.

PLANNING FOR COLLEGE. The following example should give you some idea how these options can be used together to arrive at some very meaningful figures. Let's take the case of sending your son (it could be a daughter of course) to college. If your son has just turned 9, let's assume he'll be going to college 9 years from now (108 mo) when he is 18. Let's assume that you currently have $5,000 to invest to start a fund for him and that you want to be able to send him to a college that will cost you an additional $12,000 per year at today's prices. How much will you need to send him to college and, if you started putting some money in an account, how much would you have to put in monthly to get him all the way through?

We can use Option 1 to calculate the amount of future dollars required to send him to school. Using a figure of 12000, and 7% (annual interest), for 108, 120, 132, and 144 months, we find that we need $97,950.00 in future dollars to put him through school (22061 + 23605 + 25258 + 27026). Using Option 1 again, we can determine that if we invest our current 5000 for 108 months (the beginning of school) at 11%, compounded quarterly, we will have $13,277.49 of the total needed. Subtracting this from the total, we realize that we still need $84,672.51 by the time he graduates (168 months away). Using Option 3 (Future Value) and putting in a figure of 84672.51, invested at 9.5% (.095) interest, for 168 months, with monthly frequency, we arrive at a figure of $242.76. This is the amount required per month to insure that he gets his education, assuming our estimates of interest and inflation are correct.

CAUTION. While this program is based on established formulas, other formulas are sometimes used by banks and lending institutions which could result in slightly different figures. These are to be used for planning purposes only.

```
100 REM  ******************
110 REM  * MONEY  PLANNER *
120 REM  ******************
130 REM
140 REM  AMLIST V-PL631KB
150 REM  BY T CASTLE
160 REM
170 DEF FUT=INT(((PRIN*(1+RA
TE)^PER)+.005)*100)/100
180 DEF PRE=INT(((PRIN*(1+RA
TE)^-PER)+.005)*100)/100
190 DEF FUA1=INT(((PRIN*((((
1+RATE)^PER)-1)/RATE))+.005)
*100)/100
200 DEF FUA2=(INT(((PRIN*(((
(1+RATE)^(PER+1))-1)/RATE))+
.005)*100)/100)-PRIN
210 DEF FUA3=INT(((FAMT/((((
1+RATE)^PER)-1)/RATE))+.005)
*100)/100
220 DEF PRA1=INT(((PRIN*((1-
((1+RATE)^-PER))/RATE))+.005
)*100)/100
230 DEF PRA2=(INT(((PRIN*((1
-((1+RATE)^(-PER+1)))/RATE))
+.005)*100)/100)+PRIN
240 DEF PRA3=INT(((FAMT/((1-
(1+RATE)^-PER)/RATE))+.005)*
100)/100
250 DATA FUTURE VALUE OF CUR
RENT SUM,CURRENT AMT
260 DATA COMPOUNDED-,
270 DATA PRESENT VALUE OF FU
TURE SUM,FUTURE  AMT
280 DATA COMPOUNDED-,
290 DATA FUTURE VALUE OF AN
 ANNUITY,STEADY  AMT
300 DATA FREQUENCY -,FUTURE
 SUM
310 DATA PRESENT VALUE OF AN
 ANNUITY,STEADY  AMT
320 DATA FREQUENCY -,PRESENT
 SUM
330 FOR I=1 TO 4
340 READ HEAD$(I),QUEST1$(I)
,QUEST2$(I),QUEST3$(I)
350 NEXT I
360 REM  MAIN MENU
370 CALL CLEAR
380 PRINT TAB(11);"MAIN MENU
"::
390 PRINT "  1. FUTURE VALUE
 OF CURRENT"
400 PRINT "     SUM OF MONEY
"::
410 PRINT "  2. PRESENT VALU
E OF FUTURE"
420 PRINT "     SUM OF MONEY
"::
430 PRINT "  3. FUTURE VALUE
 OF  STEADY"
440 PRINT "     PAYMENTS - A
NNUITY"::
450 PRINT "  4. PRESENT VALU
E OF STEADY"
460 PRINT "     PAYMENTS - A
NNUITY"::::
470 INPUT "  SELECTION? ":Q$
480 IF LEN(Q$)<>1 THEN 470
490 IF (ASC(Q$)<49)+(ASC(Q$)
>52)THEN 470
500 A=VAL(Q$)
510 REM  FUTURE VAL OF SUM
520 CALL CLEAR
530 PRINT HEAD$(A)
540 PRINT ::
550 IF A<3 THEN 580
560 INPUT QUEST3$(A)&"?
 ":FAMT
570 IF FAMT>0 THEN 590
580 INPUT QUEST1$(A)&"?
  ":PRIN
590 INPUT "HOW MANY MONTHS?
    ":MON
600 INPUT "ANNUAL RATE?
    ":AR
610 PRINT ::QUEST2$(A)
620 PRINT " 1- DAILY    4- Q
UARTERLY"
630 PRINT " 2- WEEKLY   5- S
EMI ANNUAL"
640 PRINT " 3- MONTHLY  6- A
NNUAL"::
650 INPUT "SELECTION?
    ":METH
```

```
660 IF METH>6 THEN 650
670 ON METH GOSUB 950,980,10
10,1040,1070,1100
680 ON A GOTO 690,720,750,83
0
690 PRINT ::"FUTURE  VALUE O
F"
700 PRINT "PRESENT   AMOUNT=
";FUT
710 GOTO 910
720 PRINT ::"PRESENT VALUE O
F"
730 PRINT "FUTURE AMOUNT    =
";PRE
740 GOTO 910
750 IF FAMT>0 THEN 800
760 PRINT ::"FUTURE VALUE OF
 "
770 PRINT "ANNUITY- ARREARS=
";FUA1
780 PRINT "          - ADVANCE=
";FUA2
790 GOTO 910
800 PRINT ::"STEADY PAYMENTS
 =";FUA3
810 FAMT=0
820 GOTO 910
830 IF FAMT>0 THEN 880
840 PRINT ::"PRESENT VALUE O
F"
850 PRINT "ANNUITY- ARREARS=
";PRA1
860 PRINT "          - ADVANCE=
";PRA2
870 GOTO 910
880 PRINT ::"STEADY PAYMENTS
=";PRA3
890 FAMT=0
900 GOTO 910
910 PRINT ::"HIT ANY KEY FOR
 MENU";
920 CALL KEY(3,KY,ST)
930 IF ST=0 THEN 920
940 GOTO 370
950 PER=MON*30
960 RATE=AR/360
970 RETURN
980 PER=(MON/12)*52
990 RATE=AR/52
1000 RETURN
1010 PER=MON
1020 RATE=AR/12
1030 RETURN
1040 PER=MON/3
1050 RATE=AR/4
1060 RETURN
1070 PER=MON/6
1080 RATE=AR/2
1090 RETURN
1100 PER=MON/12
1110 RATE=AR
1120 RETURN
```

HAPPY COMPUTING!

```
*****************
* GOLF HANDICAP *
*   V-PP831KB   *
*  BY T CASTLE  *
*****************
```

DESCRIPTION. This program should fulfil the needs of the most ambitious golf statistics enthusiast. It's a complete handicap system for not just one, but up to twelve players (probably your entire group of regulars). This is not a simple Callaway or single round handicap system. For each of the twelve players, this program stores and determines handicaps on a base of 2Ø rounds of golf. Each round, and the course rating of the course on which it was played, is stored in the array called RD$(12,2Ø). The differences are determined, the array sorted, the 1Ø largest differentials are dropped, and the handicap is calculated as .96 times the average of the remaining ten differences.

This would seem like enough for one program, but this program provides even more for the avid "hacker". Not that it's an official part of your handicap, but aren't there times when you'd like to determine how you've done on an individual hole or how many "points" you've scored? Many groups use a point system as well as handicaps when making up teams, particularly for "best ball" type play. Under this system, bogies and larger are worth zero, pars are 1, birdies are 2, etc. The number of points you score are an indication of your potential help to the team. With this program you can key in up to eight individual rounds, including your score and the par for each hole. You can use all eight for just yourself, or share the space with others. Perhaps you'd like to keep four for you and four for your spouse.

The main menu is found in lines 31Ø-47Ø and is fairly traditional from our standpoint. The normal input, save, and exit options are found as options 2, 6, and 7. Option 1 is for building and displaying the roster. For each person you'll be asked for the name and the course rating of their "home course". Although you don't input it, each time the roster is displayed, it'll also calculate and display the current handicap for each player, based on the rounds stored in memory. You can add new scores, either completed rounds, or hole by hole, using option 3. If you already have 2Ø rounds stored and you add one more, the oldest round is dropped. The same thing holds true for hole by hole, when you add the ninth round. Using the display option (5), sends you to another menu where you select either option 1 for base and handicap, or option 2 for hole by hole detail. Option 1 displays up to twenty rounds, the rating of the course, and the differential for each round. These are in chronological order from the oldest (upper left) to newest (lower right). It also shows your handicap as an integer (i.e. 14) and to one place (14.3). The display of hole by hole shows one complete round of 18 holes with the score per hole, par, and points on that hole. It also shows par for the course and your total round. The oldest round stored is shown first. After each round you're asked to "Hit any Key" and it goes to the next round and finally back to menu.

The change option on the main menu (4) is used for changing hole by hole and base handicap data, as well as for deleting an entire player. In every instance we've given you the option to "bypass" the change if you find you've made a mistake or don't understand what you're supposed to do. Bear

these things in mind. First, players must be assigned sequentially on the roster. If you have nine players on your roster and you want to remove player 3, use the change and "P" option for player (see line 35ØØ). This removes all reference to player 3, renumbers all others, including all data files and arrays pertaining to the others. Then add new players at the end of the roster. Second, if you select the change option for hole by hole or base handicap detail, you'll have to immediately replace that information with new data. Use this only if you've made a mistake on the round. This option keeps every score in it's proper sequential (chronological) order. Deleting and adding another round is not the same thing, since it takes the entire score out and adds the next one as the most recent (at the end of the array).

CAUTION. If less than 2Ø rounds are stored as base handicap detail, the computer divides the number stored in half and uses that number of rounds to calculate handicap. If it doesn't divide evenly, it goes to the next smaller integer. For instance 5 rounds would drop 3 and use 2. At seven rounds, according to our sources, this does properly calculate your "official" handicap. Our sources indicate with five rounds you use the lowest one, and at six the lowest two. Below 5 rounds you aren't supposed to have a handicap; however, we do provide some type of calculation for all rounds of 1 and up. It's the responsibility of the user to check with his/her local pro or course to determine accuracy and to update percentages and modify calculations if official rules are changed.

NOTES. This a long program and, with all of the data loaded, you have only about 95Ø bytes of memory remaining. In spite of what seems like a large amount of data, it is all "read" in from just seven 192 character data lines. We've used a number coding system and DEF statements to handle the data. These are fully described in the "Algorithms" chapter and will not be discussed here. Because of the "tight" memory situation, we've had to remove the remarks from the beginning of each subroutine. Following is the general breakout for reference:

15Ø - 29Ø	Initial Variables
31Ø - 47Ø	Main Menu
48Ø - 73Ø	Display/Change Roster
74Ø - 111Ø	Input Complete Rounds
112Ø - 174Ø	Input Hole by Hole
175Ø - 2Ø6Ø	Save Data
2Ø7Ø - 235Ø	Input Data
236Ø - 244Ø	Display Menu
245Ø - 271Ø	Handicap Base Display
272Ø - 3Ø5Ø	Hole by Hole Display
3Ø6Ø - 347Ø	Calculates Handicap
348Ø - 358Ø	Change Options
359Ø - 382Ø	Change Handicap Base
383Ø - 423Ø	Delete Player
424Ø - 457Ø	Change Hole by Hole
458Ø - 464Ø	Update Handicap Roster
NOTE: "|||"	Use FCTN A for these marks

MODIFICATIONS. For those golfers who would like to keep stats on more than twelve golfers we can make a couple of suggestions. First, you could use more than one cassette. Second, the hole by hole portions of the program do consume a good bit of the memory. With care, at this point, you should be able to remove all references to it (input lines, arrays, etc.) and expand the number of players permitted in the roster.

```
100 REM  * GOLF HANDICAP *
110 REM
120 REM  BY T CASTLE
130 REM  AMLIST V-PP831KB
140 REM
150 OPTION BASE 1
160 DIM PL$(12),CR$(12),H$(8
),RD$(12,20),ENT(18,2),VR(20
),HCP(12)
170 AD$="           "
180 ADX$="AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA"
190 FOR I=1 TO 12
200 PL$(I)=AD$
210 HCP(I)=99
220 CR$(I)="00.0"
230 FOR K=1 TO 20
240 RD$(I,K)="|||"
250 NEXT K
260 IF I>8 THEN 280
270 H$(I)=ADX$
280 NEXT I
290 DEF R$=CHR$((INT(E/3600)
)+65)&CHR$(INT((E-((INT(E/36
00)))*3600)/60)+65)&CHR$(INT
(E-((INT(E/60)))*60)+65)
300 DEF R=((ASC(SEG$(E$,1,1)
)-65)*3600)+((ASC(SEG$(E$,2,
1))-65)*60)+((ASC(SEG$(E$,3,
1))-65))
310 CALL CLEAR
320 PRINT TAB(10);"MAIN MENU
"::::
330 PRINT " 1. BUILD/DISPLAY
 ROSTER"
340 PRINT " 2. INPUT PREVIOU
S DATA"
350 PRINT " 3. ADD SCORES"
360 PRINT " 4. CHANGE SCORES
"
370 PRINT " 5. DISPLAYS"
380 PRINT " 6. SAVE DATA"
390 PRINT " 7. EXIT PROGRAM"
::::
400 INPUT "SELECTION: ":Q
410 IF (Q<1)+(Q>7)THEN 400
420 IF Q=7 THEN 450
430 ON Q GOSUB 480,2070,740,
3480,2360,1750
440 GOTO 310
450 INPUT "IS DATA SAVED (Y
OR N)?":Q$
460 IF Q$<>"Y" THEN 310
470 STOP
480 CALL CLEAR
490 PRINT "CHECKING HANDICAP
S"::
500 GOSUB 4580
510 CALL CLEAR
520 PRINT "PL#  NAME
  HCP   HCR"::
530 FOR I=1 TO 12
540 IF HCP(I)<99 THEN 570
550 HP$="NA"
560 GOTO 580
570 HP$=STR$(INT(HCP(I)+.5))
580 PRINT STR$(I);TAB(4);PL$
(I);TAB(20);HP$;TAB(25);CR$(
I)
590 NEXT I
600 PRINT ::"ENTER  PL#  TO
ADD/CHANGE OR'ZERO' TO EXIT"
::
610 INPUT "SELECTION: ":Q
620 IF Q=0 THEN 730
630 IF (Q<1)+(Q>12)THEN 610
640 CALL CLEAR
650 PRINT STR$(Q);TAB(4);PL$
(Q);TAB(25);CR$(Q)::::
660 INPUT "NAME: ":Q$
670 IF LEN(Q$)>11 THEN 660
680 PL$(Q)=SEG$(Q$&AD$,1,11)
690 PRINT
700 INPUT "RATING(72.9): ":Q
1
710 CR$(Q)=SEG$(STR$(Q1)&AD$
,1,4)
720 GOTO 480
730 RETURN
740 CALL CLEAR
750 PRINT "ENTER  (T)  TO EN
TER TOTALS FOR COMPLETE ROUN
DS OR"::
760 PRINT "ENTER  (H) FOR IN
DIVIDIUAL  HOLE BY HOLE ENTR
IES OR"::
770 PRINT "ENTER 'ZERO' TO E
XIT"::::
780 INPUT "SELECTION: ":Q$
790 IF Q$="0" THEN 1740
800 IF Q$="H" THEN 1120
810 IF Q$<>"T" THEN 780
```

```
820 CALL CLEAR
830 INPUT "PLAYER #: ":Q
840 IF Q=0 THEN 1740
850 IF (Q<1)+(Q>12)THEN 830
860 HCP(Q)=99
870 INPUT "HOME COURSE(Y OR
N)? ":Q$
880 IF Q$="Y" THEN 890 ELSE
910
890 E=VAL(CR$(Q))*10
900 GOTO 940
910 IF Q$="N" THEN 920 ELSE
870
920 INPUT "COURSE RATING(71.
5)? ":Q1
930 E=Q1*10
940 PRINT
950 INPUT "GROSS ADJ SCORE:
":Q1
960 Q1=INT(Q1)
970 E=(Q1*1000)+E
980 CK=0
990 FOR I=1 TO 20
1000 IF RD$(Q,I)="|||" THEN
1010 ELSE 1030
1010 CK=I
1020 I=20
1030 NEXT I
1040 IF CK>0 THEN 1090
1050 FOR I=1 TO 19
1060 RD$(Q,I)=RD$(Q,I+1)
1070 NEXT I
1080 CK=20
1090 RD$(Q,CK)=R$
1100 IF CGX=1 THEN 1740
1110 GOTO 740
1120 CALL CLEAR
1130 INPUT "PLAYER # OR (0)T
O EXIT: ":Q
1140 IF Q=0 THEN 740
1150 IF (Q<1)+(Q>12)THEN 113
0
1160 TENT$=SEG$(STR$(Q)&AD$,
1,2)
1170 INPUT "HOME COURSE(Y OR
 N): ":Q$
1180 IF Q$="Y" THEN 1190 ELS
E 1210
1190 TENT$=TENT$&CR$(Q)
1200 GOTO 1240
1210 IF Q$<>"N" THEN 1170
1220 INPUT "RATING (72.5): "
:Q2
1230 TENT$=TENT$&SEG$(STR$(Q
2)&AD$,1,4)
1240 CALL CLEAR
1250 PRINT "ENTER SCORE & PA
R FOR EACH  HOLE OR '0,0' TO
 EXIT"::
1260 CK=0
1270 J1=0
1280 J2=0
1290 FOR I=1 TO 18
1300 PRINT STR$(I);TAB(3);
1310 INPUT "(SCORE,PAR) ":SC
,PR
1320 IF (SC=0)+(PR=0)THEN 13
60
1330 ENT(I,1)=SC
1340 ENT(I,2)=PR
1350 GOTO 1380
1360 I=18
1370 CK=1
1380 NEXT I
1390 IF (CGX=1)*(CK=1)THEN 1
400 ELSE 1430
1400 H$(Q1)=ADX$
1410 Q$="N"
1420 GOTO 1740
1430 IF CK=1 THEN 740
1440 CALL CLEAR
1450 PRINT "HL SCR PAR
HL SCR PAR"::::
1460 FOR I=1 TO 9
1470 PRINT I;TAB(4);ENT(I,1)
;" ";ENT(I,2);TAB(16);I+9;TA
B(20);ENT(I+9,1);" ";ENT(I+9
,2)
1480 J1=J1+ENT(I,1)+ENT(I+9,
1)
1490 J2=J2+ENT(I,2)+ENT(I+9,
2)
1500 NEXT I
1510 PRINT ::::" TOTAL =";J1
;"    PAR=";J2:::
1520 INPUT "VERIFY (Y OR N):
 ":Q$
1530 IF (CGX=1)*(Q$="N")THEN
 1740
1540 IF Q$="N" THEN 740
```

```
1550 IF Q$<>"Y" THEN 1520
1560 IF CGX=1 THEN 1680
1570 FOR I=1 TO 8
1580 IF SEG$(H$(I),1,5)<>"AA
AAA" THEN 1610
1590 CK=I
1600 I=8
1610 NEXT I
1620 IF CK>0 THEN 1680
1630 FOR I=1 TO 7
1640 H$(I)=H$(I+1)
1650 NEXT I
1660 H$(8)=ADX$
1670 CK=8
1680 FOR K=1 TO 18
1690 HL$=CHR$(ENT(K,1)+65)&C
HR$(ENT(K,2)+65)
1700 TENT$=TENT$&HL$
1710 NEXT K
1720 IF CGX=1 THEN 1740
1730 H$(CK)=TENT$
1740 RETURN
1750 CALL CLEAR
1760 GOSUB 4580
1770 PRINT "REMOVE PROGRAM C
ASSETTE AND LOAD DATA CASSET
TE."::::"HIT ANY KEY"
1780 CALL KEY(3,KY,ST)
1790 IF ST=0 THEN 1780
1800 OPEN #1:"CS1",INTERNAL,
OUTPUT,FIXED 192
1810 X$=""
1820 PRINT "STORING DATA"
1830 FOR I=1 TO 12
1840 E=((VAL(CR$(I)))*10)+(I
NT(HCP(I))*1000)
1850 X$=X$&PL$(I)&R$&" "
1860 NEXT I
1870 PRINT #1:X$
1880 X$=""
1890 FOR I=1 TO 12 STEP 3
1900 FOR K=I TO I+2
1910 FOR J=1 TO 20
1920 X$=X$&RD$(K,J)
1930 NEXT J
1940 NEXT K
1950 PRINT #1:X$
1960 X$=""
1970 NEXT I
1980 FOR I=1 TO 8 STEP 4
1990 FOR K=I TO I+3
2000 X$=X$&H$(K)
2010 NEXT K
2020 PRINT #1:X$
2030 X$=""
2040 NEXT I
2050 CLOSE #1
2060 RETURN
2070 CALL CLEAR
2080 PRINT "REMOVE PROGRAM C
ASSETTE AND LOAD DATA CASSET
TE."::::"HIT ANY KEY"
2090 CALL KEY(3,KY,ST)
2100 IF ST=0 THEN 2090
2110 OPEN #1:"CS1",INTERNAL,
INPUT ,FIXED 192
2120 INPUT #1:X$
2130 FOR I=1 TO 12
2140 PL$(I)=SEG$(X$,(I*15)-1
4,11)
2150 E$=SEG$(X$,(I*15)-3,3)
2160 HCP(I)=INT(R*.001)
2170 CR$(I)=STR$((R-(HCP(I)*
1000))*.1)
2180 NEXT I
2190 FOR I=1 TO 12 STEP 3
2200 INPUT #1:X$
2210 FOR K=1 TO 3
2220 FOR J=1 TO 20
2230 RD$(I+K-1,J)=SEG$(X$,((
K*60)+(J*3))-62,3)
2240 NEXT J
2250 NEXT K
2260 NEXT I
2270 FOR I=1 TO 8 STEP 4
2280 INPUT #1:X$
2290 FOR K=1 TO 4
2300 H$(I+K-1)=SEG$(X$,(K*42
)-41,42)
2310 NEXT K
2320 NEXT I
2330 X$=""
2340 CLOSE #1
2350 RETURN
2360 CALL CLEAR
2370 PRINT TAB(9);"DISPLAY O
PTIONS"::::::
2380 PRINT "1. IND PLAYER BA
SE/HANDICAP"::::
```

```
2390 PRINT "2. IND PLAYER-RO
UND DETAIL"::::
2400 PRINT "3. MENU"::::::
2410 INPUT "SELECTION: ":Q
2420 IF Q=3 THEN 3050
2430 IF Q=2 THEN 2720
2440 IF Q<>1 THEN 2410
2450 CALL CLEAR
2460 INPUT "PLAYER #: ":Q
2470 IF (Q<1)+(Q>12)THEN 246
0
2480 GOSUB 3060
2490 PRINT "RAW  RTG DIFF  R
AW  RTG DIFF":::
2500 FOR I=1 TO 10
2510 J=0
2520 E$=RD$(Q,I+J)
2530 E1$=STR$(INT(R/1000))
2540 E2$=STR$((R-(VAL(E1$)*1
000))/10)
2550 IF VAL(E1$)<215 THEN 26
00
2560 E3$="0"
2570 E1$="0"
2580 E2$="0"
2590 GOTO 2610
2600 E3$=STR$(VAL(E1$)-VAL(E
2$))
2610 IF J=10 THEN 2650
2620 PRINT E1$;TAB(5);E2$;TA
B(10);E3$;
2630 J=10
2640 GOTO 2520
2650 PRINT TAB(16);E1$;TAB(2
0);E2$;TAB(25);E3$
2660 NEXT I
2670 IF CGX=1 THEN 3050
2680 PRINT ::"HANDICAP ";INT
(HCP(Q)+.5);" (";INT((HCP(Q)
+.05)*10)/10;")"
2690 PRINT ::"HIT ANY KEY";
2700 CALL KEY(3,KY,ST)
2710 IF ST=0 THEN 2700 ELSE
3050
2720 CALL CLEAR
2730 INPUT "PLAYER #: ":Q
2740 PRINT
2750 IF (Q<1)+(Q>12)THEN 273
0
2760 FOR I=1 TO 8
2770 RESTORE 2780
2780 DATA 0,0,0,0,0,0
2790 READ E1,E2,E3,E4,E5,E6
2800 IF SEG$(H$(I),1,2)="AA"
 THEN 3040
2810 IF VAL(SEG$(H$(I),1,2))
<>Q THEN 3040
2820 CALL CLEAR
2830 PRINT "HL SC PR PT    H
L SC PR PT"::
2840 FOR K=1 TO 9
2850 J=0
2860 E1=ASC(SEG$(H$(I),((K+J
)*2)+5,1))-65
2870 E2=ASC(SEG$(H$(I),((K+J
)*2)+6,1))-65
2880 E3=(E2+1)-E1
2890 IF E3>0 THEN 2910
2900 E3=0
2910 E4=E4+E1
2920 E5=E5+E2
2930 E6=E6+E3
2940 IF J=9 THEN 2980
2950 PRINT STR$(K+J);TAB(4);
STR$(E1);TAB(7);STR$(E2);TAB
(10);STR$(E3);
2960 J=J+9
2970 GOTO 2860
2980 PRINT TAB(16);STR$(K+J)
;TAB(19);STR$(E1);TAB(22);ST
R$(E2);TAB(25);STR$(E3)
2990 NEXT K
3000 PRINT ::"SCORE: ";E4:"P
AR:   ";E5:"POINTS:";E6::::
3010 PRINT "HIT ANY KEY"
3020 CALL KEY(3,KY,ST)
3030 IF ST=0 THEN 3020
3040 NEXT I
3050 RETURN
3060 J=0
3070 FOR I=1 TO 20
3080 IF RD$(Q,I)="|||" THEN
3140
3090 E$=RD$(Q,I)
3100 E1=INT(R/1000)
3110 E2=(R-(E1*1000))/10
3120 J=J+1
3130 GOTO 3160
3140 VR(I)=-99
3150 GOTO 3170
```

```
3160 VR(I)=E1-E2
3170 NEXT I
3180 DF=15
3190 IF DF=0 THEN 3320
3200 FOR I=1 TO 20-DF
3210 FG1=I
3220 FG2=FG1+DF
3230 IF VR(FG1)>=VR(FG2)THEN
 3290
3240 HD=VR(FG1)
3250 VR(FG1)=VR(FG2)
3260 VR(FG2)=HD
3270 FG1=FG1-DF
3280 IF FG1>0 THEN 3220
3290 NEXT I
3300 DF=INT(.5*DF)
3310 GOTO 3190
3320 IF J>1 THEN 3380
3330 IF J=1 THEN 3350
3340 GOTO 3460
3350 J1=1
3360 J2=1
3370 GOTO 3400
3380 J1=INT((.5*J)+.5)+1
3390 J2=INT(.5*J)
3400 HP=0
3410 FOR I=J1 TO J
3420 HP=HP+VR(I)
3430 NEXT I
3440 HCP(Q)=(HP/J2)*.96
3450 IF HCP(Q)>0 THEN 3470
3460 HCP(Q)=0
3470 RETURN
3480 CALL CLEAR
3490 PRINT "TO CHANGE OR DEL
ETE"::
3500 PRINT "ENTER P FOR PLAY
ER"::
3510 PRINT "ENTER R FOR ROUN
DS"::
3520 PRINT "ENTER H FOR HOLE
 BY HOLE    "::
3530 INPUT "SELECTION: ":Q$
3540 IF (Q$<>"R")*(Q$<>"H")*
(Q$<>"P")THEN 3480
3550 INPUT "PLAYER #:  ":Q
3560 IF (Q<1)+(Q>12)THEN 355
0
3570 IF Q$="H" THEN 4240
3580 IF Q$="P" THEN 3830
3590 CALL CLEAR
3600 CGX=1
3610 HCP(Q)=99
3620 PRINT "COUNT TOP TO BOT
TOM"::
3630 PRINT "1-10 ON LEFT  11
-20 ON RIGHT":::
3640 GOSUB 2500
3650 PRINT
3660 INPUT "WHICH RND? ":Q1
3670 IF (Q1<1)+(Q1>20)THEN 3
660
3680 PRINT "D TO DELETE OR"
3690 PRINT "C TO CHANGE   "
3700 INPUT "B TO BYPASS   ":
Q1$
3710 IF Q1$="B" THEN 4560
3720 IF Q1$="D" THEN 3780
3730 IF Q1$<>"C" THEN 3680
3740 RD$(Q,Q1)="|||"
3750 CALL CLEAR
3760 GOSUB 870
3770 GOTO 4560
3780 FOR I=Q1 TO 19
3790 RD$(Q,I)=RD$(Q,I+1)
3800 NEXT I
3810 RD$(Q,20)="|||"
3820 GOTO 4560
3830 CALL CLEAR
3840 PRINT "THIS DELETES ALL
 DATA"
3850 PRINT "FOR PLAYER";Q;PL
$(Q)::::
3860 PRINT "ENTER (D) TO DEL
ETE"
3870 INPUT "       (B) TO BYP
ASS    ":Q1$
3880 IF Q1$="B" THEN 4560
3890 IF Q1$<>"D" THEN 3830
3900 FOR I=Q TO 11
3910 PL$(I)=PL$(I+1)
3920 CR$(I)=CR$(I+1)
3930 HCP(I)=HCP(I+1)
3940 FOR K=1 TO 20
3950 RD$(I,K)=RD$(I+1,K)
3960 NEXT K
3970 NEXT I
3980 PL$(12)=AD$
3990 HCP(12)=99
4000 CR$(12)="0"
```

```
4010 FOR K=1 TO 20
4020 RD$(12,K)="|||"
4030 NEXT K
4040 FOR I=1 TO 8
4050 IF SEG$(H$(I),1,2)="AA"
 THEN 4120
4060 J=VAL(SEG$(H$(I),1,2))
4070 IF J<Q THEN 4120
4080 IF J>Q THEN 4110
4090 H$(I)=ADX$
4100 GOTO 4120
4110 H$(I)=SEG$(STR$(J-1)&AD
$,1,2)&SEG$(H$(I),3,40)
4120 NEXT I
4130 J=0
4140 FOR I=1 TO 8
4150 HOLD$=H$(I)
4160 IF HOLD$=ADX$ THEN 4190
4170 J=J+1
4180 H$(J)=HOLD$
4190 NEXT I
4200 FOR I=J+1 TO 8
4210 H$(I)=ADX$
4220 NEXT I
4230 GOTO 4560
4240 CALL CLEAR
4250 CGX=1
4260 PRINT "OLDEST ROUND LIS
TED FIRST"::
4270 FOR I=1 TO 8
4280 IF SEG$(H$(I),1,2)="AA"
 THEN 4350
4290 IF VAL(SEG$(H$(I),1,2))
<>Q THEN 4350
4300 T=0
4310 FOR K=1 TO 18
4320 T=T+(ASC(SEG$(H$(I),(K*
2)+5,1))-65)
4330 NEXT K
4340 PRINT "RND: ";I;TAB(10)
;"SCORE: ";T
4350 NEXT I
4360 IF T=0 THEN 4560
4370 PRINT ::
4380 INPUT "WHICH ROUND? ":Q
1
4390 IF (Q1<1)+(Q1>8)THEN 43
80
4400 PRINT "D TO DELETE OR"
4410 PRINT "C TO CHANGE"
4420 INPUT "B TO BYPASS    ":
Q1$
4430 IF Q1$="B" THEN 4560
4440 IF Q1$="D" THEN 4520
4450 IF Q1$<>"C" THEN 4400
4460 H$(Q1)=ADX$
4470 CALL CLEAR
4480 GOSUB 1160
4490 IF (CGX=1)*(Q$="N")THEN
 4560
4500 H$(Q1)=TENT$
4510 GOTO 4560
4520 FOR I=Q1 TO 7
4530 H$(I)=H$(I+1)
4540 NEXT I
4550 H$(8)=ADX$
4560 CGX=0
4570 RETURN
4580 FOR UD=1 TO 12
4590 IF HCP(UD)<99 THEN 4630
4600 IF RD$(UD,1)="|||" THEN
 4630
4610 Q=UD
4620 GOSUB 3060
4630 NEXT UD
4640 RETURN
```

HAPPY COMPUTING!

CHAPTER TWELVE

Summary and Looking Ahead

GENERAL. You've now reached the end of the instructional material in this manual. (Note, we didn't say you've reached the end of the learning process.) Whether it's taken you three months, six months, or even a year or more to complete this, if you've performed even a minimum number of the experiments and if you've entered all of the programs, you should now be able to comfortably sit down in front of your computer feeling that you are totally in command of this very useful tool. By now you should also recognize that there are certain basic principles that we've tried to convey to you.

The first of these is that this relatively small 16K computer, equipped with only a single cassette recorder, is not a toy. It is capable of storing, manipulating, and displaying a wide variety of complex data. You've worked with major programs that have proven this. Now consider what you can do to help yourself. Do you collect stamps or rare coins? Are you into family trees or Biorythms? Did you ever lend out a library book and forget it? Do you need a cross reference for a tape or record collection? Have you ever wondered what difference it would make in your power bill if you reduced the size of your light bulbs from 100 to 75 or 60 watt bulbs, or if you changed them all to fluorescent. The computer excels at these kinds of projects. If you have young children you certainly know the value from an educational standpoint. Comparison and multiple choice type programs are easy to write using the random statements, and they don't have to be real fancy to beat the old hand written "flash card". States and capitals and presidents of the United States are popular favorites. More and more books are appearing daily with programs you can enter yourself; you can modify programs designed for other computers; you can create your own; or buy them on cassette or in module form. It's a tool waiting to be used.

Second, you should now understand how a computer "thinks". That's probably a bad term, because a computer doesn't really think at all, it just compares variables and obeys your commands. It does things in exactly the order you specify, without ever considering whether it's logical or not. It can't accept "gray" areas, just yes or no, right or wrong. In order to get a computer to perform a complex task, you had to break it down into very small little tasks and explain in detail how to complete each one. When you begin to think about every potential computer application in this manner, you'll be well on your way to being an expert programmer. If you keep breaking the problem down into smaller parts, there will be very few tasks that you can't complete.

Third, there are certain techniques that can actually make your computer more powerful. By now you should recognize these and know where to find examples of their usage when in doubt. Rather than just tell you that string arrays save memory, we gave you numerous test routines to prove the theory. You've learned the value of the subroutine approach and how this

helps keep in perspective the specific problem you're working on. Once these problems are worked out the solutions are yours forever -- available for use anytime. We also hope, in conjunction with this, that you've learned the value of EXPERIMENTING with your computer. Speed of execution and memory consumption are the key factors in data processing, regardless of what size system you operate or how many peripherals you have. Each command has its own personality -- its strengths and its weaknesses. To determine who's going to play what position on a football or baseball team, the coach has TRYOUTS. To find the proper actor or actress for a play they ask people to READ for them. How can you decide which command or which series of commands is the best if you haven't looked at each one carefully and compared it to the others? If you move on to Extended Basic, disk drive, mini memory, etc., we hope you've learned how to develop your own EXPERIMENTS to determine the capabilities of each and every new command available.

The fourth thing that we hope you've come to understand is that the "error message" is the programmer's best friend. If you get a lot of error messages, it doesn't mean that you're not a good programmer. It could mean you're using your time wisely or trying new things. On the one hand, you'll probably enter code 100% faster if you don't bother to check your spelling before hitting the ENTER key. If it's wrong, the computer will tell you either immediately or when you run it that you have an error. It takes far less time to correct the few you will spell incorrectly than it does to avoid them in the first place. On the other hand, when you experiment, you'll always get some error messages. They say that Thomas Edison tried thousands of different materials as a filament for his light bulb before he found the right one. Every time he tried one and failed, he got an "error message". He didn't look upon these errors as failures, he figured he was successful in discovering a material that wouldn't work. The more error messages you see, the more proficient you'll become at programming, because, the next time, you'll know all the things that don't work.

Finally, by now you should have a feel for how much programming you want to do. It is certainly exhilarating to take a concept and develop it from an idea to a completed running program that does what you want it to do. However, complex programs are not just "jotted off", but may take weeks to fully write and verify. Perhaps you simply don't have that much free time. Only you can weigh the relative value of time of development versus the cost of purchasing commercially prepared programs. However, if you do decide to buy, what you now know about programming should enable you to make much wiser decisions. If you're able to review such a program before purchasing, you can now decide whether it can be modified to meet the specific need that you have.

Converting Other Programs. A tremendous number of educational and useful programs are published monthly in specialized magazines devoted to the TI and in other magazines and books for other computers. Obviously, those programs written specifically for your systems, as it is set up, with or without disk, printer, etc., are the best source. However, when it comes to converting others, care must be

exercised. Assuming all are written in some form of BASIC, there are some clues you can look for to help you determine how difficult or simple the conversion might be.

If you're trying to convert to straight console basic and the base program utilizes multiple statements on a single program line it will complicate the problem. While not impossible, bear in mind that almost all of the line number references in GOTO, GOSUB, IF statements, etc., will be different unless you take precautions while entering. If you want to attempt it, don't skip ten lines between each of your entries and renumber it as you go. Until you've completed entering the whole program, keep your basic line number the same as the main line number for the multiple statement. Take a look at the following two multiple line statements:

```
>480 DISPLAY AT(14,6):"REPLAY
 ? PRESS REDO" :: DISPLAY AT
 (16,4):"TO END QUIT"
>490 CALL KEY(0,A,B)::IF J=10
  THEN 700 ELSE 900
```

Enter these as follows:

```
>480 DISPLAY AT(14,6):"REPLAY
 ? PRESS REDO"
>485 DISPLAY AT(16,4):"TO END
  QUIT"
>490 CALL KEY(0,A,B)
>495 IF J=10 THEN 700 ELSE 900
```

If you entered these as 480, 490, 500, and 510, by the time it got to 700 or 900 you could be hundreds off on your line reference.

The second thing to consider is whether the program is heavily involved in graphics or sound. The methods used by the new computers differ greatly between manufacturers as to how they access these capabilities, if they have them at all. Aside from giving you a concept for a program, they probably won't convert very well on a line for line basis. Look in computer book stores for some of the older books written for micro computers. In many ways these can be a better source than the newer programs. Many of these were written before graphics, sound, and color were available. A football or baseball game may simply move an "X" or an "*" around the screen. A lunar lander may be nothing more than an "A". Still, all of the other statements that actually control the game, like IF, GOTO, etc., may be perfectly alright. After the program is running, you can go back to the beginning and redefine characters and add sound and color statements to liven it up.

In the standard statements like PRINT and INPUT, you'll need to analyze the separators used after the statement and what they mean for each system. Generally, it's just a change in the punctuation mark. If you see statements like the following, with numbers following print or input statements, the program requires direct screen placement of messages.

```
>120 POSITION 2,10:? "AGAIN? ":Q
```

While we can create a subroutine that more or less will do this in Console Basic, it would be far easier in Extended Basic. The method used for formatting numbers varies also. Some use a statement like PRINT USING or PRINT %#10F2, etc. Extended Basic has the capability of doing most anything other similar size systems will do, while console basic is somewhat limited in its ability. Another major area of

concern is the method used to reference sections of string variables. Console basic uses the SEG$ command and requires a start point and number of characters. Others use LEFT$ (left string), RIGHT$(right string), or MID$(middle string), and may specify the starting and ending characters or how many characters. A program that requires a lot of string handling may be difficult to convert. Lastly, consider how much memory it requires. Many published programs tell you how many bytes they consume or what type of system they were created on. If not, be sure to look over the opening lines for DIM statements before you start. Do a little quick math to determine how much memory they'll consume.

Of course, we're going to make the assumption that you've enjoyed your data processing experience thus far, and that you want to pursue it further. How do you select from the hundreds of peripherals and add-ons available today? Everyone's needs differ, but we'd like to discuss the relative merits of several of the more popular expansions and give you our opinion of what they have to offer.

Extended Basic. This is a relatively inexpensive addition to the straight console basic unit and is well worth the investment. With its additional commands and capabilities you'll be able to accomplish considerably more with it than with comparably priced units. For creating games and educational type programs, the addition of moving sprites is practically essential. The ability to specify positions for PRINT and INPUT statements, to format numbers, validate on input, and combine commands on a single line, are tremendous assets for serious applications. For user friendly programs, the ON ERROR capability is fantastic. With proper use of this command you can virtually eliminate the problem of an error bringing a program to a halt. It's not our purpose to go into all of the additional commands available, but they are extensive. By all means, "Buy It!"

Speach Synthesizer. This is a very popular peripheral. For purely functional programs like mailing lists, accounting, amortization schedules, etc., this would be more of a "gimmick" than a real necessity; however, for games and educational programs for children, it is probably one of the greatest things ever invented. The impact of being able to make the computer "come alive" and talk to a child can't be overstated.

Printer/Peripheral Expansion. There is no doubt that the addition of a printer, and the required Peripheral Expansion unit to interface it, is the next most important add-on. In addition to aiding tremendously in the development and debugging of programs, you would then have the capability of getting hard copy printouts of things like checkbook data, statistical data, etc. The most difficult decision you'll have is deciding what type of printer to buy. For home use, the choice basically comes down to either a dot-matrix type printer or a daisy-wheel type. Regardless of what manufacturer you go with, a dot-matrix type printer will normally offer greater flexibility in computer controlled type sizes and it operates at a faster speed; however, the copy produced will always look like it came off a computer. The daisy-wheel type gives you good letter quality (just as good as you would get off the best

typewriter) but it operates somewhat slower and usually doesn't permit computer controlled changes in type style. For true word processing like sales letters, books, etc., the daisy-wheel is a must. If you're limiting your printing to programs, labels, and reports, perhaps the dot-matrix will do.

Disk Drive. Beyond the printer, certainly a disk drive would be the next most important peripheral. This will open up a whole new world of possibilities because of its speed and the ability to store multiple programs and data files on a single medium. The flexibility and random access capability afforded you by being able to use RELATIVE files in conjunction with the RECord statement, and the APPEND and UPDATE modes, mean that you're no longer limited to just what you can hold in memory. Compared to a cassette recorder, relatively huge data files can be sorted and searched in a "wink". Don't forget that you're still going to need "backup" copies of all your programs and data. On cassettes, we suggested you always make at least two copies of a program or data file. If you only have one disk drive, although it may hold 10 or 20 programs, you're going to need a lot of individual cassettes to back it up. If you don't back it up, and just one disk is destroyed, you may lose all of those programs at one time. If you intend to get into disk drive for something like permanent accounting records, to be realistic, you'll have to consider buying two disk drives. With two drives you won't have to make extra copies one program at a time or one data file at a time. An entire disk can be copied from one drive to another with one command.

Additional Memory. If you have disk drive, for many applications you'll eliminate the need for additional memory. Few programs actually consume 16K in themselves -- it's the data they load and manipulate that consumes the memory. With disk drive, you don't always need to load all of the data into memory to manipulate it; therefore, you don't need as much memory. There are three applications where extra memory is important. First, if you're going to do a lot of word processing, 16K will probably not be sufficient. Text adds up fast and, in order to edit and make changes to documents, you'll want to be able to load the entire copy into memory. A normal printed page may contain as many as 3000 characters. If your operating programs consume 6 or 7 thousand bytes, you won't be able to work on more than 1 or 2 pages at a time. Extra memory is important for word processing. Heavy sorting applications or programs which do a lot of mathematical calculations may also require additional memory. If you're sorting a large number of names and addresses (1000 or more), even with a disk drive, just loading the ZIP code and record number into memory may require more than the 16K capability. Programs that do a lot of "number crunching", calculating possibilities and probabilities, like programs for chess games or some card games (like bridge) require large amounts of data in memory.

Modems. Modems, those little devices that permit you to communicate with other computers via telephone lines, can open up a whole new world of possibilities. More on-line services are being created every day which permit you to: obtain stock reports and current news; access thousands of programs, store large amounts of data

in mainframe computers; obtain vast amounts of information from already established data bases; and communicate with other personal computers. The cost of the initial installation is relatively small and the "on-line" charges are nominal considering the capabilities.

Third Party Peripherals. As each new micro computer enters the market, so do scores of third party manufacturers of peripherals. We're referring more to the hardware now than the software. You've probably already seen numerous configurations of disks drives, printers, modems, etc., manufactured by companies other than Texas Instruments. In many cases it will seem, and perhaps it's even so, that these manufacturers can offer greater capabilities at lower prices. Will it be a wise investment? Before we answer this question, understand that we're not affiliated with TI or any other manufacturer, nor do we sell hardware ourselves; therefore, we have no vested interest in promoting or discouraging any particular purchase. When making your decision, you should ask yourself two questions. First, "What do I know about the company?" If the peripheral is purely solid state electronics, made up of nothing but "chips", "diodes", etc., once operational, it may never fail (or not in your lifetime). If it has any moving parts, such as a printer, disk drive, etc., in all probability it will at some point require service. Who's going to perform that service. Second, "How much do I know about hardware?" Many people, armed with their basic "Owners Manual" are capabable of tuning their own cars. The specifications are clearly spelled out for timing, dwell, angle, etc., and all you need to do is turn a couple of screws to get it set properly. This holds true as long as you keep the car in factory condition. If you add a special carburetor, disconnect polution devices, or add a "blower", you can forget about what the manuals say. If you add peripherals, other than those manufactured by TI, the rules regarding what you may or may not be able to do, both now and in the future, may no longer apply. If you have enough knowledge to figure out the new rules on your own, third party peripherals may afford you some advantages.

IN CONCLUSION

In order to get the maximum out of your TI-99/4A, or any other computer you may be involved with in the future, we have two parting words of advice. First, read everything you can that pertains directly to your system, and read about other systems as well. Read the "Letters to the Editor", as well as instructional sections. You're sure to find one or two little subroutines or tricks in every one of these publications at least once a year. That one small idea may save 10-20 hours of work on your part and more than justifies the expense. Second, keep experimenting. You've heard it time and time again, but it bears repeating. You'll never know the limits of a system until you experiment with it to the point of "failure". When you get an error message, try to figure out another way around the problem and just keep going forward. With these final words, we wish you --

HAPPY COMPUTING!

```
**************************
*     KAMAKAZE  RUN      *
*       V-PE331KB        *
**************************
```

DESCRIPTION. Don't be deceived by the size of this program. It contains an abundance of graphics, sounds, and movement. As the program begins running, the player is given a black screen with two rows of Kamakaze planes at the top. There are twelve planes in each row. The bottom of the screen has a green band with three white buildings to the left and a hovering, blue gunship to fend of the Kamakaze pilots. Superimposed on the green band is a white zero to the left. This number will be replaced later by the high score during each session of play. The white zero to the right is where the current score will be recorded as the game progresses. In the center of the screen there is a message instructing the player to "HIT ANY KEY".

After hitting a key, the lowest row of pilots (red) begin to drop, randomly, toward the ground. If they are permitted to drop to a level just above the blue gunship, they will begin a bombing run across the screen to the left and bomb the first building still standing. You control the movement of the blue gunship with the left and right arrow and fire using the "period". When enough of the red planes have started dropping, the yellow planes begin dropping. They will not come below the level of the highest red plane. Once they start their bomb run, you cannot shoot the plane down. If you lose all three buildings the game is over and the opening display is again put up. If you clear the board by shooting down all 24 planes (red and yellow) your buildings are rebuilt and 24 more planes are again placed at the top. With each successive board, the planes drop more quickly and the score for each kill is increased. You can pause at any time by hitting the "P". A "PAUSE" message is placed on the screen and you can begin again by hitting any key. The values of the red planes on levels 1, 2, and 3 are 50, 100, 150 points respectively. The value of the yellow planes on levels 1, 2, and 3, are 150, 225, and 300 points respectively. Consider yourself fortunate if you get get to the fourth board and a score of 20,000 points or over.

SEQUENCE. This program is layed out very much like the discussion of game programs in Part 2. The general sequence is found in lines 100-300 and the main subroutines are as follows: initial variables lines 310-620; starting display lines 630-970; movement of row 2 and branching statement to bomb run (line 1130) lines 980-1270; movement of row 1 and branching statement to bomb run (line 1410) lines 1280-1710; gunship movement and scoring routine lines 1720-2240. Additional subroutines for bomb attack, printing score, printing high score, and printing messages are found at 2250, 2930, 3010, and 3090. The variable LVL controls how many rows each plane drops on each movement. Scoring is based on the LVL value and is found in lines 2060 and 2150.

This program is primarily made possible by setting up two arrays (line 330, R1 and R2) which keep track of the current row location for each of the 24 planes which are dropping.

BASIC TIPS BY AMLIST
Published by: AMLIST, Inc.
4542 Memorial Dr. #202
Decatur, GA 30032
Editor: Terrance K. Castle (404) 292-0576

```
100 REM  ****************
110 REM  * KAMIKAZE RUN *
120 REM  ****************
130 REM  BY T CASTLE
140 REM  AMLIST V-PE331KB
150 REM
160 GOSUB 320
170 GOSUB 640
180 GOSUB 1730
190 IF EN=3 THEN 160
200 IF MX1+MX2=24 THEN 170
210 GOSUB 990
220 IF EN=3 THEN 160
230 IF MX1+MX2=24 THEN 170
240 GOSUB 1730
250 IF EN=3 THEN 160
260 IF MX1+MX2=24 THEN 170
270 GOSUB 1290
280 IF EN=3 THEN 160
290 IF MX1+MX2=24 THEN 170
300 GOTO 180
310 REM  SET START VALUES
320 CALL CLEAR
330 DIM R1(12),R2(12)
340 IF SCR>HSCR THEN 370 ELSE 350
350 HSCR=HSCR
360 GOTO 380
370 HSCR=SCR
380 EN=0
390 SCR=0
400 LVL=2
410 CALL SCREEN(2)
420 CALL COLOR(3,16,13)
430 CALL COLOR(4,16,13)
440 CALL COLOR(5,16,1)
450 CALL COLOR(6,16,1)
460 CALL COLOR(7,16,1)
470 CALL COLOR(8,16,1)
480 CALL COLOR(13,11,1)
490 CALL COLOR(14,7,1)
500 CALL COLOR(15,13,1)
510 CALL COLOR(16,16,1)
520 CALL COLOR(12,5,1)
530 E1$="007E7E3C3C181800"
540 CALL CHAR(128,E1$)
550 CALL CHAR(136,E1$)
560 CALL CHAR(137,"1084200A80240000")
570 CALL CHAR(144,"FFFFFFFFFFFFFFFF")
580 CALL CHAR(152,"0F0909FFFF9999FF")
590 CALL CHAR(120,"18187E7EFFFF7E00")
600 CALL CHAR(153,"0000181818180000")
610 CALL CHAR(154,"0000000000006FFF")
620 RETURN
630 REM  START DISPLAY
640 LVL=LVL+1
650 FOR I=1 TO 12
660 R1(I)=2
670 R2(I)=1
680 NEXT I
690 MX1=0
700 MX2=0
710 EN=0
720 L1=1
730 L2=13
740 L3=1
750 L4=13
760 P1=16
770 FOR I=1 TO 22
780 CALL HCHAR(I,1,32,32)
790 NEXT I
800 FOR J=5 TO 27 STEP 2
810 CALL HCHAR(1,J,128)
820 CALL HCHAR(2,J+1,136)
830 NEXT J
840 CALL HCHAR(24,1,144,32)
850 GOSUB 2940
860 GOSUB 3020
870 CALL HCHAR(23,4,152)
880 CALL HCHAR(23,6,152)
890 CALL HCHAR(23,8,152)
900 CALL HCHAR(22,P1,120)
910 IF SCR>0 THEN 970
920 MSG$="1210HIT ANY KEY"
930 GOSUB 3100
940 CALL KEY(3,KY,ST)
950 IF ST=0 THEN 940
960 CALL HCHAR(12,10,32,12)
970 RETURN
980 REM  MOVES ROW 2
990 IF MX2=12 THEN 1270
1000 RANDOMIZE
1010 T1=0
1020 IF R2(L3)=25 THEN 1030 ELSE 1050
1030 L3=L3+1
1040 GOTO 1060
1050 L3=L3
1060 IF R2(L4-1)=25 THEN 1070 ELSE 1090
1070 L4=L4-1
1080 GOTO 1100
1090 L4=L4
1100 M1=INT((L4-L3)*RND)+L3
1110 IF R2(M1)>=25 THEN 1270
```

```
1120 IF R2(M1)>21-LVL THEN 1130 ELSE 1170
1130 GOSUB 1620
1140 IF EN=3 THEN 1270
1150 R2(M1)=25
1160 GOTO 1260
1170 FOR T=1 TO 12
1180 IF R1(T)<=R2(M1)+LVL THEN 1190 ELSE 1210
1190 T1=1
1200 T=12
1210 NEXT T
1220 IF T1=1 THEN 1270
1230 CALL SOUND(500,-4,0)
1240 CALL HCHAR(R2(M1)+LVL,(M1*2)+3,128)
1250 CALL HCHAR(R2(M1),(M1*2)+3,32)
1260 R2(M1)=R2(M1)+LVL
1270 RETURN
1280 REM  MOVES ROW 1
1290 IF MX1=12 THEN 1490
1300 IF R1(L1)=25 THEN 1310 ELSE 1330
1310 L1=L1+1
1320 GOTO 1340
1330 L1=L1
1340 IF R1(L2-1)=25 THEN 1350 ELSE 1370
1350 L2=L2-1
1360 GOTO 1380
1370 L2=L2
1380 M1=INT((L2-L1)*RND)+L1
1390 IF R1(M1)>=25 THEN 1490
1400 IF R1(M1)>21-LVL THEN 1410 ELSE 1450
1410 GOSUB 1510
1420 IF EN=3 THEN 1490
1430 R1(M1)=25
1440 GOTO 1490
1450 CALL SOUND(500,-4,0)
1460 CALL HCHAR(R1(M1)+LVL,(M1*2)+4,136)
1470 CALL HCHAR(R1(M1),(M1*2)+4,32)
1480 R1(M1)=R1(M1)+LVL
1490 RETURN
1500 REM    BOMB RUN 1
1510 CALL VCHAR(20-LVL,(M1*2)+4,32,LVL+2)
1520 FOR I=(M1*2)+4 TO 8 STEP -1
1530 CALL SOUND(200,-4,0)
1540 CALL HCHAR(21,I,136)
1550 CALL HCHAR(21,I,32)
1560 NEXT I
1570 MX1=MX1+1
1580 VL1=136
1590 GOSUB 2260
1600 RETURN
1610 REM  BOMB RUN 2
1620 CALL VCHAR(20-LVL,(M1*2)+3,32,LVL+2)
1630 FOR I=(M1*2)+3 TO 8 STEP -1
1640 CALL SOUND(200,-4,0)
1650 CALL HCHAR(21,I,128)
1660 CALL HCHAR(21,I,32)
1670 NEXT I
1680 MX2=MX2+1
1690 VL1=128
1700 GOSUB 2260
1710 RETURN
1720 REM  GUN MOVEMENT
1730 CALL KEY(3,KY,ST)
1740 IF ST=0 THEN 2240
1750 IF KY=80 THEN 1790
1760 IF KY=83 THEN 1860
1770 IF KY=68 THEN 1910
1780 IF KY=46 THEN 1960 ELSE 2240
1790 MSG$="1213PAUSE"
1800 GOSUB 3100
1810 CALL KEY(3,KY,ST)
1820 IF ST=0 THEN 1810
1830 MSG$="1213      "
1840 GOSUB 3100
1850 GOTO 1730
1860 IF P1-1<5 THEN 2240
1870 P1=P1-1
1880 CALL HCHAR(22,P1+1,32)
1890 CALL HCHAR(22,P1,120)
1900 GOTO 2240
1910 IF P1+1>28 THEN 2240
1920 P1=P1+1
1930 CALL HCHAR(22,P1-1,32)
1940 CALL HCHAR(22,P1,120)
1950 GOTO 2240
1960 G1$=STR$((P1/2)-2)
1970 IF VAL(G1$)<1 THEN 1990
1980 IF LEN(G1$)<3 THEN 2080
1990 G2=VAL(G1$)+.5
2000 G1=R2(G2)
2010 IF G1=25 THEN 2020 ELSE 2040
2020 G1=1
2030 GOTO 2070
2040 R2(G2)=25
2050 MX2=MX2+1
2060 SCR=SCR+((LVL-1)*75)
2070 GOTO 2160
2080 G2=VAL(G1$)
2090 G1=R1(G2)
2100 IF G1=25 THEN 2110 ELSE 2130
2110 G1=1
2120 GOTO 2160
2130 R1(G2)=25
2140 MX1=MX1+1
2150 SCR=SCR+((LVL-2)*50)
```

```
2160 FOR G=21 TO G1 STEP -2
2170 CALL SOUND(50,-3,0)
2180 CALL HCHAR(G,P1,153)
2190 CALL HCHAR(G,P1,32)
2200 NEXT G
2210 CALL HCHAR(G1,P1,32)
2220 CALL SOUND(300,-5,0,120,0)
2230 GOSUB 2940
2240 RETURN
2250 REM  BOMB ATTACK 1
2260 CALL GCHAR(23,8,BL1)
2270 CALL GCHAR(22,8,TST)
2280 IF TST=120 THEN 2290 ELSE 2300
2290 EN=2
2300 IF BL1=152 THEN 2570
2310 FOR I=7 TO 6 STEP -1
2320 CALL SOUND(200,-4,0)
2330 CALL HCHAR(21,I,VL1)
2340 CALL HCHAR(21,I,32)
2350 NEXT I
2360 CALL GCHAR(23,6,BL1)
2370 CALL GCHAR(22,4,TST)
2380 IF TST=120 THEN 2390 ELSE 2400
2390 EN=2
2400 IF BL1=152 THEN 2670
2410 FOR I=5 TO 4 STEP -1
2420 CALL SOUND(200,-4,0)
2430 CALL HCHAR(21,I,VL1)
2440 CALL HCHAR(21,I,32)
2450 NEXT I
2460 CALL GCHAR(23,4,BL1)
2470 CALL GCHAR(22,4,TST)
2480 IF TST=120 THEN 2490 ELSE 2500
2490 EN=2
2500 IF BL1=152 THEN 2770
2510 FOR I=3 TO 1 STEP -1
2520 CALL SOUND(200,-4,0)
2530 CALL HCHAR(21,I,VL1)
2540 CALL HCHAR(21,I,32)
2550 NEXT I
2560 GOTO 2920
2570 CALL HCHAR(22,8,153)
2580 CALL HCHAR(22,8,32)
2590 CALL HCHAR(23,8,153)
2600 CALL HCHAR(23,8,137)
2610 CALL SOUND(200,-5,0,170,0)
2620 CALL SOUND(450,-5,0,120,0)
2630 CALL HCHAR(23,8,154)
2640 EN=EN+1
2650 J=7
2660 GOTO 2870
2670 CALL HCHAR(22,6,153)
2680 CALL HCHAR(22,6,32)
2690 CALL HCHAR(23,6,153)
2700 CALL HCHAR(23,6,137)
2710 CALL SOUND(200,-5,0,170,0)
2720 CALL SOUND(450,-5,0,120,0)
2730 CALL HCHAR(23,6,154)
2740 EN=EN+1
2750 J=5
2760 GOTO 2870
2770 CALL HCHAR(22,4,153)
2780 CALL HCHAR(22,4,32)
2790 CALL HCHAR(23,4,153)
2800 CALL HCHAR(23,4,137)
2810 CALL SOUND(200,-5,0,170,0)
2820 CALL SOUND(450,-5,0,120,0)
2830 CALL HCHAR(23,4,154)
2840 EN=EN+1
2850 J=3
2860 GOTO 2870
2870 FOR I=J TO 1 STEP -1
2880 CALL SOUND(200,-4,0)
2890 CALL HCHAR(21,I,VL1)
2900 CALL HCHAR(21,I,32)
2910 NEXT I
2920 RETURN
2930 REM  PRINT SCORE
2940 MS$=STR$(SCR)
2950 L=LEN(MS$)
2960 FOR I=1 TO L
2970 MS=ASC(SEG$(MS$,I,1))
2980 CALL HCHAR(24,20+I,MS)
2990 NEXT I
3000 RETURN
3010 REM  PRINT HI SCORE
3020 MS$=STR$(HSCR)
3030 L=LEN(MS$)
3040 FOR I=1 TO L
3050 MS=ASC(SEG$(MS$,I,1))
3060 CALL HCHAR(24,8+I,MS)
3070 NEXT I
3080 RETURN
3090 REM  PRINT ANY MSG
3100 MSR=VAL(SEG$(MSG$,1,2))
3110 MSC=VAL(SEG$(MSG$,3,2))
3120 L=LEN(MSG$)-4
3130 MS$=SEG$(MSG$,5,L+4)
3140 FOR I=1 TO L
3150 MS=ASC(SEG$(MS$,I,1))
3160 CALL HCHAR(MSR,MSC+I,MS)
3170 NEXT I
3180 RETURN
```

MAJOR TOPICS

*STRUCTURING PROGRAMS

*CREATING FORMULAS

*DEBUGGING

*GRAPHICS

*ARRAYS

*CONDENSING & REFINING

*VALIDITY & TESTING

*DATA FILES

*SORTING

*SOUND

*CONSERVING MEMORY

16 MAJOR PROGRAMS — INCLUDING

*PERSONAL CHECKBOOK PROGRAM W/CHART OF ACCOUNTS & BAR GRAPH DISPLAY PROGRAM FOR BUDGET VS ACTUAL

*SPORTS PROGRAMS FOR GOLF, BASEBALL, & BOWLING

*CALENDER SCHEDULING PROGRAM FOR MAJOR EVENTS

*EDUCATIONAL PROGRAMS FOR YOUNG CHILDREN

*GAME PROGRAMS FOR THE FAMILY

*MONEY PLANNING PROGRAM

*FIVE PROGRAMS CREATE AND USE DATA FILES

ALL PROGRAMS & EXAMPLES REQUIRE ONLY CONSOLE BASIC & ONE RECORDER